Expert: Otimizando a performance de aplicações VCL Win32 usando técnicas avançadas de computação

Clube Delphi +PHP

DevMedia group

Ano 9 :: Edição 110 :: R$ 11,90

ISSN 1517990-7

Ask the Expert
Generics no Delphi Win32

QuickUpdate
Firebird, AJAX e Prism

Web - Novidades
Curso de Delphi Prism
Classes parciais, tipos anônimos e métodos de extensão

Easy Delphi
Programação Orientada a Objetos – Parte 3
Polimorfismo na prática!

Delphi for PHP
Consumindo Web Services

Boas Práticas e POO

Crie sistemas mais flexíveis e com baixo custo de manutenção usando:
- Programação para Interfaces vs. Herança
- Polimorfismo
- Padrões de Projeto

Brinde
Comprando essa edição ganhe
4 vídeos

- Dominando a OO e Componentes – Gerência de Memória – Parte 1
- Dominando a OO e Componentes – Gerência de Memória – Parte 2
- Dominando a OO e Componentes – UML
- Novidades no Delphi 2009 – Ribbon Controls

SISFÁCIL 2.0

FICOU AINDA, MAIS FÁCIL ADMINISTRAR SEU SITE

Enquete, Fotos, Notícias, Blogs e muito mais.

DISPONÍVEL PARA CLIENTES RJHOST

Seu site pronto e completo

www.sisfacil.com.br

RJHOST

Sumário

Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

Quick Update

9 – Firebird, AJAX e Prism
[*Guinther Pauli*]

Delphi

Expert

13 – Otimizando funções da VCL
Aumente a performance de funções, como o FieldByName, utilizando listas hash
[*Thiago Borges de Oliveira*]

Novidades

22 – Curso de Delphi Prism – Parte 3
Novidades da Linguagem – Classes parciais, Tipos anônimos e Métodos de extensão
[*Guinther Pauli*]

Boas Práticas

Boas Práticas

28 – POO – Crie sistemas flexíveis, expansíveis e com baixo custo de manutenção
Programando para interfaces, usando polimorfismo e padrões de projetos
[*Rafael Stavarengo*]

Easy Delphi

Easy Delphi, Tutorial

37 – Introdução à POO – Parte 3
Aprenda, de uma forma simples, a programar da forma correta
[*Guinther Pauli*]

PHP

Delphi for PHP

43 – Web Services – Parte 2
Crie clientes em Win32, PHP e ASP.NET para consumir Web Services do Delphi PHP
[*Rodrigo Carreiro*]

Você percebeu os ícones ao lado de cada matéria? Eles indicam o que você vai encontrar no artigo – dessa forma, você também pode ter uma idéia geral do que vai encontrar nesta edição como um todo! Os editores trabalham sempre no sentido de fechar a revista seguindo esta definição, para oferecer a você o melhor conteúdo didático!

Confira abaixo a lista com a definição dos tipos de artigo encontrados nesta edição:

[Quick Update] Artigos que atualizam o desenvolvedor sobre determinada tecnologia, seja ela nova ou não.

[Novidades] Artigos sobre tecnologias de ponta, que ainda não fazem parte o dia-a-dia do desenvolvedor mas que prometem ser a próxima febre.

[Expert] Artigo com foco no leitor avançado.

[Boas Práticas] Um dos objetivos da revista é levar para o leitor não somente as melhores técnicas mas também as melhores práticas. Esse tipo de artigo foca em técnicas que poderão aumentar a qualidade do desenvolvimento de software.

[Easy Delphi] Com foco no desenvolvedor iniciante.

[Tutorial] Artigo no estilo tutorial passo a passo.

[Delphi for PHP] Artigo com foco na versão Delphi for PHP.

Clube Delphi +PHP

Ano 9 - 110ª Edição 2009 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Sobrinho
daniel.sobrinho@devmedia.com.br

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Diagramação e Capa
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther.pauli@devmedia.com.br

EDITORIAL

Em primeiro lugar, é um enorme prazer estar de volta. Em segundo lugar, o Delphi 2010 está aí com muitas novidades: suporte ao Windows 7, Multi-Touch, driver dbExpress nativo para Firebird (finalmente), RTTI melhorada, DataSnap 2010 com suporte a REST (JSON) e muitas novidades no IDE. Para minha alegria, e acredito para a maioria dos que estão lendo isto, a paleta de componentes estilo Delphi 7 está de volta. Temos agora um "Googlezinho" dentro do IDE, apertando F6, será aberto o IDE Insight, recurso que localiza praticamente qualquer coisa no ambiente: componentes, code templates, opções de projeto, opções de environment, debugger etc. Compilação em background, formatador automático de código, novos componentes, enfim, muitas novidades interessantes, que você poderá conferir já nas próximas edições aqui da ClubeDelphi.

Como se já não bastasse, o Delphi Prism (que eu prefiro chamar de D#) está a todo o vapor. Nesta edição trouxe dois artigos, a terceira parte do curso que mostra as novidades da linguagem, onde apresento os novos recursos relacionados ao uso de classes: classes parciais, classes/tipos anônimos, classes/métodos de extensão e classes com construtores estendidos (object initializer). No QuickUpdate, um super upgrade para você conferir em poucos minutos o que há de mais interessante no mundo do desenvolvimento Web com o Prism: ASP.NET 3.5 e AJAX com Firebird. E vem mais pela frente, em breve aplicações RIA com ADO.NET Data Services, REST, AJAX e Silverlight.

Boas Práticas: nossa matéria capa vai para um belo artigo sobre Programação Orientada a Objetos, do Rafael, onde mostra como modelar um framework usando boas práticas, como programação orientada a interfaces e não implementação (herança), dar prioridade à composição e separar o que muda. Resultado: sistemas mais flexíveis, expansíveis e adaptáveis a mudanças.

Para os Experts: o Thiago mostra algo impressionante de ser feito com o Delphi, injetar código na memória da aplicação em execução para desviar chamadas a funcionalidades da VCL. E ainda por cima, em uma aplicação prática, aumentar a performance do sistema. Para os iniciantes, aprenda tudo sobre métodos virtuais, abstratos, propriedades, construtores, destrutores, especificadores de visibilidade e muito mais na terceira parte do curso de OO. Para finalizar, na sessão PHP, o Rodrigo mostra como consumir Web Services.

Boa Leitura!

Guinther Pauli
guinther.pauli@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix
MCP,MCAD,MCSD.NET,MCTS,MCPD
guintherpauli.blogspot.com
twitter.com/guintherpauli

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

ClubeDelphi PLUS

Informativo ClubeDelphi

Portal ClubeDelphi

www.clubedelphi.net/portal

+ 700 vídeo aulas e 19 cursos online

Brinde na web desta edição - 4 vídeos! Confira:

4 Vídeos

Dominando a orientação a objetos e componentes (avançado): Gerenciamento de memória e casting – Parte 1
Entenda como o Delphi trabalha ao alocar memória para um objeto.
http://www.devmedia.com.br/articles/viewcomp.asp?comp=13614

Dominando a orientação a objetos e componentes (avançado): Gerenciamento de memória e casting – Parte 2
Entenda com Rodrigo Carreiro Mourão o que é casting e como utilizar os operadores is e as.
http://www.devmedia.com.br/articles/viewcomp.asp?comp=13615

Dominando a orientação a objetos e componentes (avançado): UML
Veja como criar diagramas UML dentro do Delphi, utilizando o Together Live Source
http://www.devmedia.com.br/articles/viewcomp.asp?comp=13907

Novidades no Delphi 2009 - Ribbon Controls
Utilize no Delphi a nova estrutura de menus Ribbon Controls, estilo Office 2007
http://www.devmedia.com.br/articles/viewcomp.asp?comp=12218

Para acessar os vídeos utilize os seguintes dados

Login: **DVM.PLS** *Senha:* **xkm23v**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Edições Anteriores da ClubeDelphi

Você pode comprar todas as edições anteriores da ClubeDelphi através do site da DevMedia!

Para isso basta acessar *https://seguro.devmedia.com.br/edicoes_anteriores.asp*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PLS** Senha: **xkm23v**

Perguntas e Respostas

Generics no Win32

Perguntas respondidas por Guinther Pauli
guinther.pauli@gmail.com

Prezados amigos da Revista ClubeDelphi. Sou leitor da revista desde a edição 51. Há anos tenho lido muito sobre *Generics*, agora vi que o suporte está incluído no *Delphi Win32*. Desenvolvo em *Delphi Win32* há anos e gostaria de saber como posso usar esse recurso em minhas aplicações atuais com *Delphi 2010*.

Alisson Freitas Dornelles
Balneário Camboriu - SC

Generics não é um conceito do *Delphi* e já existe em outras linguagens como C# e *Java*. Este recurso já era suportado no *Delphi for .NET* desde o *Delphi 2007* e também agora no *Delphi Prism*. Com o lançamento do *Delphi 2009*, e continuando no *Delphi 2010*, esse brilhante recurso foi incluído na linguagem *Delphi Win32*. *Generics* podem ser usados de diferentes formas, especialmente prefiro usar para coleções, mas não se limita a isso. Podemos usar *Generics* na definição de tipos, métodos, parâmetros etc. Basicamente, um *Generic* é um parâmetro, que ao invés de ser um valor, é um tipo. Em muitas situações, ele é usado onde colocaríamos um *TObject*, indicando que nossa classe, método, coleção, pode trabalhar com qualquer tipo. Com *Generics*, não precisamos especificar o tipo momento da definição da funcionalidade, da estrutura, da nova classe, mas sim no momento que vamos usá-la. E o melhor de tudo, *Generics* são rápidos e evitam e muito o uso de RTTI. Vamos a alguns exemplos.

Na **Listagem 1**, temos um exemplo simples de como usar *Generics* com tipos. Declaramos um *array of <alguma coisa, um tipo>*. Depois, no momento em que realmente vamos usar o *array*, o declaramos e definimos então o seu tipo. No exemplo, defini dois *arrays*, um de nomes de pessoas e outro de cores.

Na **Listagem 2** temos um exemplo mais elaborado. Eu adaptei para o *Delphi Win32* um exemplo clássico da documentação da MSDN sobre *Generics* com C#. O método *Troca* (*Swap*) inverte os valores de duas variáveis. Só que na declaração do método e na sua implementação, não trabalhamos com um tipo pré-definido, ou um

Listagem 1. Generics em tipos

```
type
  TMeuArray<T> = array of T;
...
procedure TForm1.TestandoMeuArray();
var
  Nomes: TMeuArray<String>;
  Cores: TMeuArray<TColor>;
  I: integer;
begin
  SetLength(Nomes,2);
  Nomes[0] := 'Ben Campbell';
  Nomes[1] := 'Micky Rosa';
  for I := 0 to Length(Nomes) - 1 do
     ShowMessage(Nomes[i]);
  SetLength(Cores,2);
  Cores[0] := clRed;
  Cores[1] := clBlue;
  Randomize();
  Color := Cores[Random(1)];
end;
```

TObject, trabalhamos com qualquer coisa, um *T*, um genérico. No real momento da utilização do método, dizemos ao método *Troca* que tipo de dado vamos usar. Com isso, podemos usar o método *Troca* para trocar valores de inteiros, strings, chars etc.

No meu ponto de vista, o melhor uso de *Generics* é com coleções (**Listagem 3**), é onde ganhamos mais velocidade. Antes, no *Delphi*, tínhamos coleções básicas, como *ObjectList*. O problema dessas coleções é que elas poderiam armazenar qualquer coisa dentro, ou seja, trabalham diretamente com *TObject*, ou na pior das hipóteses, *pointers*. A coleção não vai ser fortemente tipada. Se quisermos usar uma lista para colocar somente objetos do tipo *TClientDataSet*, podemos usar a mesma coleção para colocar botões, *edits* etc., porque

Listagem 2. Usando Generics em métodos

```
procedure TForm1.Troca<T>(var Obj1,Obj2: T);
var
  Temp: T;
begin
  Temp := Obj1;
  Obj1 := Obj2;
  Obj2 := Temp;
end;
...
procedure TForm1.TestandoTroca();
var
  X,Y: integer;
  S1,S2: string;
begin
  X := 1;
  Y := 2;
  Troca<Integer>(X,Y);
  S1 := 'Ben';
  S2 := 'Micky';
  Troca<String>(S1,S2);
end;
```

obviamente ela aceita qualquer *TObject*. E outra, no momento de usar o objeto, retirando-o da lista, precisamos fazer *type-cast*, e por tabela, isso implica usar RTTI, que é lenta. Usando *Generics*, o compilador gera uma coleção fortemente tipada no momento da compilação, que aceita somente tipos do parâmetro que você definir, sem a necessidade de *type-casts* e conversões. Note que nossa coleção é parametrizada, para aceitar somente objetos do tipo *Pessoa*.

Dúvida respondida pelo editor Guinther Pauli. Envie suas dúvidas para o email: guinther.pauli@gmail.com ●

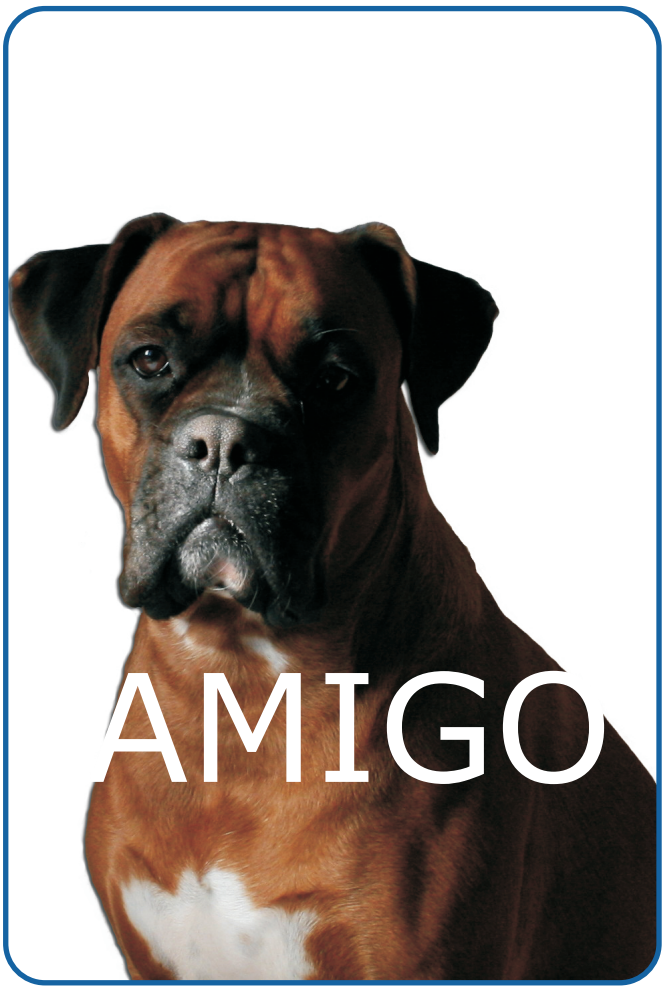
Listagem 3. Generic em coleções

```
// uses Generics.Collections;
type
  Pessoa = class
  private
    FIdade: integer;
    FNome: string;
  public
    constructor Create(ANome: string; AIdade: integer);
  published
    property Nome: string read FNome write FNome;
    property Idade: integer read FIdade write FIdade;
  end;
...
constructor Pessoa.Create(ANome: string; AIdade: integer);
begin
  Nome := ANome;
  Idade := AIdade;
end;
...
procedure TForm1.TestandoCollection();
var
  Pessoas: TList<Pessoa>;
begin
  Pessoas := TList<Pessoa>.Create();
  Pessoas.Add(Pessoa.Create('Guinther',31));
  Pessoas.Add(Pessoa.Create('Ben',21));
  // Pessoas.Add(Form1); <- não funciona
end;
```

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback



Existem coisas
que não conseguimos
ficar sem!

...só pra lembrar,
sua assinatura pode
estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

Para mais informações:
www.devmedia.com.br/central

DevMedia GROUP

Quick Update: Firebird, AJAX e Prism

Guinther Pauli
guinther.pauli@gmail.com
Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software, dos quais 14 dedicados ao Delphi e 9 ao .NET/C#. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Delphi para Linux, Certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET, VB.NET, MSF, SQL Server). Editor geral da Revista ClubeDelphi, Editor Geral da Revista .NET Magazine, professor de Pós-Graduação. Publicou mais de 150 artigos, 400 vídeo-aulas, palestrante em 5 edições da Borland Conference. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adotação de melhores práticas. Blog: guintherpauli.blogspot.com

O desenvolvimento de aplicações Web com o *Delphi* está finalmente consolidado, depois de tantas idas e vindas. A tecnologia que usamos hoje é *Delphi Prism*, *Visual Studio 2008*, *ASP.NET 3.5 + ASP.NET AJAX*, e futuramente, *Delphi Prism*, *Visual Studio 2010*, ASP.NET AJAX 4.0, *ADO.NET Data Services* e *ADO.NET Entity Framework*.

O banco de dados utilizado pela maioria dos desenvolvedores Delphi é, sem dúvida, o *Firebird*, apesar do *SQL Server* ser o banco mais indicado para trabalhar com o *ASP.NET*. Então, já que estamos na coluna *QuickUpdate*, que tal uma atualização bem rápida, tipo uma reciclagem em 10 min., uma aplicação de cadastro de usuários que utilize as tecnologias disponíveis atualmente no *Prism*? Vamos lá.

Antes de começar, é importante saber o que faremos: uma aplicação Web multicamadas, onde o browser enviará as informações digitadas de um cadastro de usuários (*Nome* e *Email*) via AJAX, ou seja, uma chamada assíncrona XML-HTTP sem *postbacks* e sem *refreshs* de página. Essas informações serão recebidas por um *Web Service*, que repassará os dados para uma camada a parte de acesso ao *Firebird* (chamada DAL), que usa *ADO.NET* e o *provider* nativo do *Firebird* para enviar dados a uma *Stored Procedure*.

Criando o banco de dados

Primeiro, crie no *Firebird* um banco de dados chamado *USUARIOS.FDB* e dentro dele uma tabela com a estrutura da **Figura 1.** Como você pode ver, é uma tabela extremamente simples. Nosso

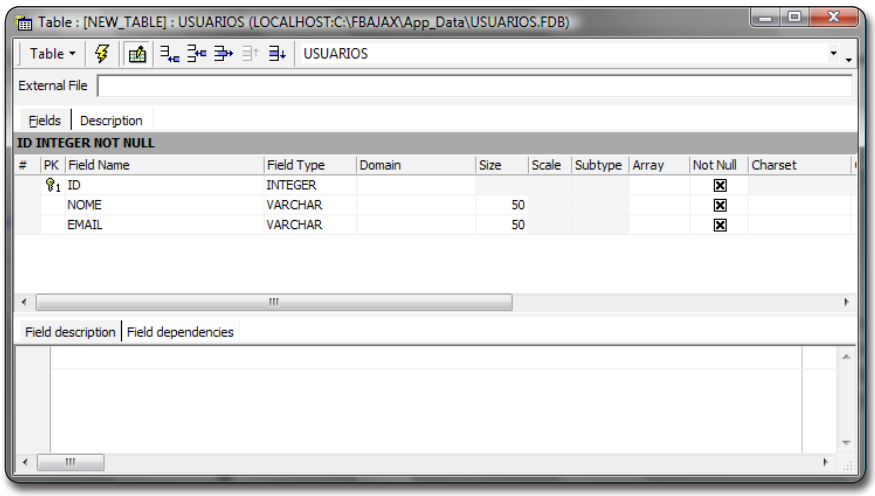


Figura 1. Tabela de usuários

foco aqui não é o cadastro em si, mas as tecnologias envolvidas. Estou usando o *IBExpert* para criar o BD, logo, usei a opção *AutoInc* para o ID usando *Sequence/Generator* e *Trigger*. Criei também uma *Stored Procedure* para inserir os dados nessa tabela, como mostra a **Listagem 1.**

Criando o Web Site

No *Prism*, inicie um novo *ASP.NET Web Site*, escolhendo a linguagem *Oxygene*. No formulário principal, bem no topo, coloque um *ScriptManager* da aba *AJAX Extensions*. Esse componente será responsável por gerenciar as chamadas assíncronas feitas à pagina. No form principal, coloque dois *TextBox's* (digite "Nome" e "Email" acima deles) e um *Input Button* (HTML). Seu formulário deve estar semelhante ao mostrado na **Figura 2.**

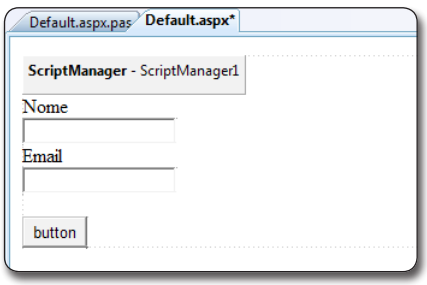


Figura 2. Formulário principal da aplicação

Implementando o acesso a dados

Agora precisamos implementar o acesso a dados, nesse caso, vamos trabalhar apenas com a operação de inserção. Para acesso ao *Firebird* a partir do *ASP.NET*, vamos usar o que há de melhor, o driver do próprio *Firebird* para *ADO.NET*. Então baixe e instale-o a partir do endereço *http://www.firebirdsql.org/index.php?op=files&id=netprovider* (versão para *.NET 3.5*). Jamais use um *SqlDataSource* ou *dbxDataSource* em aplicações reais.

No *ASP.NET* vamos implementar nossa classe de acesso a dados, chamada DAL (*Data Access Layer*). Clique de direita no nome do *Web Site* no *Solution Explorer* e escolha *Add>New Item>Class*, dando o nome de DAL para ela (coloque na *App_Code* quando questionado). O código é mostrado na **Listagem 2.** Para

Listagem 1. Stored Procedure para inserir dados na tabela

```
create procedure InsertUsuario (
    NOME varchar(50),
    EMAIL varchar(50))
as
begin
  insert into USUARIOS(NOME,EMAIL)
  values (:NOME,:EMAIL);
end;
```

poder usar os objetos do *provider ADO.NET* de acesso ao *Firebird*, clique de direita novamente na *Solution* e localize (*browse*) o arquivo *FirebirdSql.Data.FirebirdClient.dll*, por padrão em *c:\Program Files\FirebirdClient*. Aqui basicamente usamos *ADO.NET* e seus objetos para enviar os dados ao BD. *FbConnection* faz a conexão, enquanto o *FbCommand* executa a instrução através do seu método *ExecuteNonQuery*. Tudo com parâmetros, para evitar ataques de *SQL Injection*.

Criando o Web Service

O próximo passo é criar o *Web Service* que receberá os dados do browser via AJAX. Para isso, no *Solution Explorer* dê um clique de direita e escolha *Add>New Item>Web Service* e implemente-o conforme a **Listagem 3.** Aqui nada de mais, simplesmente recebemos os parâmetros e repassamos à classe DAL. Poderíamos ter feito o acesso a dados aqui no próprio *Web Service*, mas separar o *ADO.NET* em uma camada a parte nos dá mais independência, além de ser uma boa prática.

Chamando o Web Service via JavaScript (AJAX)

No formulário, selecione o *ScriptManager* e abra o editor da sua propriedade *Services*. Adicione um novo item, definindo o *InlineScript* para *True* e *Path* para *~/WebService.asmx*. Agora,

Nota

Não vou me preocupar com o layout aqui, o VS2008 possui um excelente novo recurso para editar estilos CSS. É recomendável, inclusive, que você trate o layout da aplicação somente no final do desenvolvimento.

Listagem 2. Classe de acesso ao Firebird usando o provider ADO.NET

```
namespace;

interface

uses
  System,
  FirebirdSql.Data.FirebirdClient;

type
  DAL = public class
  public
    method Insert(Nome, Email: String);
    // aqui iriam os demais métodos, para update, select etc
  end;

implementation

method DAL.Insert(Nome, Email: String);
begin
  // Coloque na string abaixo o caminho do seu banco
  // Quando em produção, coloque a string no Web.Config
  var ConStr := "User=SYSDBA;Password=masterkey;Database=
   C:\FBAJAX\App_Data\Usuarios.fdb;DataSource=localhost;";
  var Con := new FbConnection(ConStr);
  // caso não queira usar SPs use o comando abaixo
  // var SQL := "insert into USUARIOS(Nome,Email) values
   (@Nome,@Email)";
  var SQL := "InsertUsuario";
  var Cmd := new FbCommand(SQL,Con);
  // Só use a atribuição abaixo quando usar SPs
  Cmd.CommandType := System.Data.CommandType.StoredProcedure;
  Cmd.Parameters.AddWithValue("@Nome", Nome);
  Cmd.Parameters.AddWithValue("@Email", Email);
  Con.Open();
  try
    Cmd.ExecuteNonQuery();
  finally
    Con.Close();
  end;
end;

end.
```

Listagem 3. Web Service para receber dados do browser

```
namespace;

interface

uses
  System,
  System.Web.Services,
  System.Web.Services.Protocols;

type
  [WebService(&Namespace := 'http://tempuri.org/')]
  [WebServiceBinding(ConformsTo := WsiProfiles.BasicProfile1_1)]
  // É necessário decorar a classe com o atributo abaixo
  // para poder chamar a partir de uma app AJAX
  [System.Web.Script.Services.ScriptService]
  WebService = public class(System.Web.Services.WebService)
  public
    method Insert(Nome, Email: String);
  end;

implementation

[WebMethod]
method WebService.Insert(Nome, Email: String);
begin
    var NewDAL := new DAL();
    NewDAL.Insert(Nome,Email);
end;

end.
```

abra o arquivo ASPX e adicione o código da **Listagem 4**, dentro do elemento *Head*. Note que esse código vai rodar no **navegador,** ou seja, é uma chamada assíncrona que não se comunica com a página, não faz *postback* e gera o mínimo de tráfego necessário. O que fizemos aqui foi adicionar um *eventhandler* ao botão HTML para chamar o método *Insert*, que aciona o *Web Service* e passa os parâmetros. Dois métodos de *Callback* dão o feedback ao usuário, no caso de sucesso ou falha.

Listagem 4. Código JavaScript usa AJAX para invocar o Web Service

```
<script type="text/javascript">
function pageLoad()
{
    $addHandler( $get("Button1"), "click", Insert );
}
function Insert() {
     WebService.Insert(
        $get("TextBox1").value,
        $get("TextBox2").value,
        Sucesso,Erro);
}
function Sucesso() {
    alert("Usuário inserido com sucesso");
}
function Erro() {
    alert("Erro ao inserir usuário");
}
</script>
```

Testando

Pronto! Pressione *Ctrl+F5* e teste a aplicação no browser (**Figura 3**).

Algumas dicas importantes caso haja falha na aplicação:

1 - Primeiro teste sua *Stored Procedure* no banco, via IBExpert;

2 - Depois, teste o *Web Service* e a classe *DAL*, dando um clique de direita no arquivo *ASMX* no *Solution Explorer* e escolhendo *View in Browser*;

3 - Finalmente, teste a aplicação no browser, lembrando de setar a *default.aspx* como página padrão (clique de direita + *Set as Start Page*). Uma dica final, considere o uso do IE para teste da sua aplicação AJAX, ele possui uma melhor integração para exibição de erros e depuração.

Simples? Sim, com certeza, mas veja que algumas pequenas coisas fazem a diferença:

1 – Usamos *Stored Procedures*, o que aumenta em muito a performance de aplicações, principalmente em ambiente Web *multi-thread*;

2 – Desenvolvemos uma aplicação multicamadas, uma excelente prática em *ASP.NET*. Uma dica interessante seria criar mais uma camada (de negócio, chamada *BLL*) e objetos *DTO* para trocas de informações entre camadas (veja meu artigo na edição 98);

Dica

Note que você pode usar o IntelliSense (o CodeInsight do Visual Studio) para digitar seu código JavaScript.

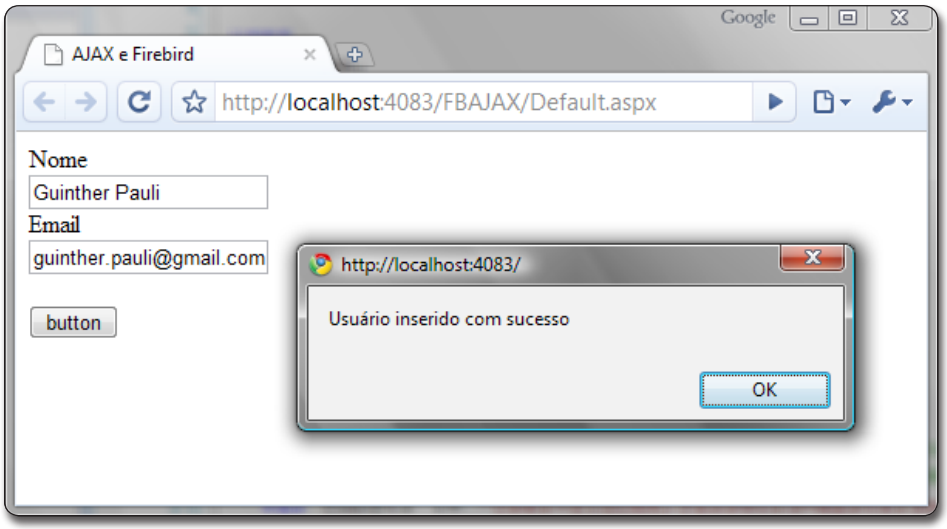


Figura 3. Testando a aplicação AJAX inserindo dados no Firebird

3 – Usamos o *provider* nativo do *Firebird* para *ADO.NET*, o que aumenta a performance consideravelmente, principalmente se compararmos ao uso de outros *providers*, como *OleDB*, *ODBC*, *BDP*;

4 – Ao usar o *provider* nativo do *Firebird*, por tabela contamos com um *Connection Pooling*, que basicamente é uma *cache* de conexões em memória. Ou seja, um *Open* não necessariamente vai abrir uma conexão, pode pegar uma do *pool*, o que literalmente faz a aplicação voar;

5 – E finalmente, enviamos a tela para o browser e não trafegamos mais nada além de dados, via chamadas a um *Web Service*, usando *JavaScript* e AJAX. Essa é uma tendência forte no desenvolvimento de aplicações Web hoje, focar no desenvolvimento CLIENT.

Futuro?

O *Visual Studio 2010* e *.NET 4.0* já estão aí em versão beta, o *Delphi Prism* certamente estará lá quando for lançada a versão final (já existe uma versão beta do *Prism* para *VS2010*). O *ASP.NET 4.0* vem focado fortemente no desenvolvimento *client-side* e *JavaScript*. É isso mesmo, depois de anos e anos acreditando que deveríamos colocar todo o processamento no servidor, seja em uma aplicação client/server, em SPs no BD, no servidor Web, ou no *app server* em multicamadas, o futuro traça outro rumo. Uma tecnologia chamada *ADO.NET Data Services* usa o padrão REST que permite acesso aos dados como recursos, e não serviços, através da URL. Isso roda no servidor, com acesso a dados feito com *ADO.NET Entity Framework*. O browser não se comunica com um *Web Form ASP.NET*, mas usa AJAX e muito processamento cliente para receber dados em XML ou JSON e exibir ao cliente. Mais? Sim, vamos usar *Silverlight* para desenvolver aplicações RIA (*Rich Internet Applications*) com Prism e obter dados do *ADO.NET Data Services*. Em suma, acabou o vai e vem de telas na Web, enviamos o form (seja ASP.NET, Silverlight, Flash, HTML) uma única vez e a seguir toda a comunicação é assíncrona. Se usar ASP.NET AJAX 4.0, contará com o uso de *client-templates*, que são basicamente, a grosso modo, uma versão cliente em *JavaScript* de controles servidores ASP.NET. Tudo em *HTML*, *CSS* e *JavaScript*. E o *Delphi 2010 (Win32)* não vai ficar de fora não, por exemplo, o *DataSnap 2010* também suportará padrões REST e JSON. Está preparado? A ClubeDelphi está. O *Prism* certamente estará.

Bem-vindos ao futuro. Grande abraço e até a próxima edição. ●

Links

See Delphi Prism in Visual Studio 2010 Beta 1
http://blogs.remobjects.com/blogs/mh/2009/05/19/p282

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Otimizando funções da VCL

Aumente a performance de funções, como o FieldByName, utilizando listas hash

Do que trata o artigo

O artigo descreve como é feita a alteração de comportamento de classes da VCL em tempo de execução. A alteração será feita através de substituição de código diretamente nas páginas de memória, durante a inicialização do aplicativo. Alterar o código de partes compiladas da VCL obriga a posterior recompilação de componentes de terceiros e partes da própria VCL.

Para que serve

Esta técnica é utilizada para substituir funções que são executados com muita frequência e que se substituídas por versões otimizadas, podem trazer uma grande melhoria no desempenho ou tempo de execução de aplicativos. Considera-se que não se pode alterar o código por vias normais, por exemplo, estendendo a funcionalidade de uma classe através de herança.

Em que situação o tema é útil

Estas substituições de código são frequentemente utilizadas pelos projetos DelphiSpeedUp e FastCode. Como eles substituem partes da RTL por códigos otimizados, a fim de melhorar o tempo de execução da IDE, são obrigados a fazer as alterações em memória, já que não podem modificar o código fonte da IDE e VCL. As técnicas podem ser aplicadas no dia a dia para otimizar partes da VCL e estruturas de dados utilizadas nos aplicativos.

Resumo do DevMan

Este artigo mostra como se pode alterar o comportamento de um aplicativo em tempo de execução, substituindo partes de código na memória por outras, otimizadas. Esta técnica é usada pelos desenvolvedores do projeto FastCode, um projeto que tem a intenção de reescrever várias funções da VCL do Delphi com o objetivo de torná-las mais rápidas. O projeto tem uma visibilidade tal que a própria CodeGear/Embarcadero tem atualizado algumas de suas funções com base no que foi disponibilizado no projeto. Além disso, esta técnica também é utilizada por Andreas Hausladen em seus utilitários para Delphi. Um exemplo de uso da técnica é o utilitário DephiSpeedUp que substitui funções do Delphi e torna a IDE muito mais rápida. Mais informações sobre o plug-in podem ser obtidas aqui: *http://andy.jgknet.de/blog/?page_id=8*

Thiago Borges de Oliveira
thborges@gmail.com
Mestrando em Sistemas Distribuídos na UFG, pesquisando R-Trees e Joins distribuídos. Desenvolvedor de drivers DBX4 para Firebird, Postgres, SQLite e Oracle. Arquiteto/Analista de software, programando em Delphi/C++/Java/Python e automatizando teste de software com TestComplete.

Desenvolver um *framework* como a VCL não é uma tarefa fácil. Ter em mente todos os possíveis usos que um desenvolvedor poderá fazer de uma quantidade imensa de classes não é tão simples. Devido a estas dificuldades, muitas partes do código não recebem a devida atenção ou não são codificadas pensando em requisitos agressivos de desempenho. Muitas vezes você nem sabe que usa, mas estes códigos estão lá no seu executável final, rodando no cliente e levando um tempo precioso para ser processado.

Além do mais, há a questão do uso de conjuntos de instruções dos processadores mais novos. Do que adianta comprar um processador com um conjunto de instruções *SSE3* se o compilador não gera executáveis que fazem uso delas?

Mas existe um bom motivo para evitar o uso destes conjuntos de instruções: O executável tem que ser compatível com todos os processadores. Mas não teria uma forma de aproveitar tais instruções quando disponíveis?

Pensando nestes detalhes, o projeto *FastCode* cria rotinas substitutas para funções que são muito chamadas durante a execução do programa. Tais rotinas identificam as capacidades do processador da máquina e substituem funções da RTL e VCL por versões otimizadas, adequadas ao processador em uso.

O interessante da abordagem que o projeto *FastCode* usa é que basta apenas incluir a *unit* com as otimizações no seu projeto. Nada de alterações no código de seu aplicativo para usufruir das otimizações. Mas como isso é feito? Como substituir funções da VCL sem alterar os *pas/dcus* da mesma? Ora, o compilador lê estes arquivos ao gerar o executável, não é mesmo?

Esta substituição não é algo simples. De fato seria muito mais fácil alterar o código da *unit* da VCL diretamente. Mas isso implica em recompilar todas as dependências da *unit* alterada, inclusive componentes de terceiros e até mesmo partes da própria VCL. O impacto da alteração é grande e se espalha bastante, dependendo da *unit* alterada. Já recebeu a mensagem de erro na compilação: "*Unit x was compiled with a different version of y*"?

Para entender o que é feito, voltemos à arquitetura de computadores que usamos: a arquitetura de *Von Neuman*. Nela, todo programa sai do dispositivo de armazenamento secundário (disco rígido) e vai para a memória principal para ser processado. E é aqui que mora o segredo: páginas de memória podem ser alteradas.

Existem métodos no sistema operacional que permitem a alteração de partes da memória enquanto o programa está sendo executado. Isto é feito a todo o tempo, na área de dados do aplicativo, quando o valor de variáveis é alterado, quando dados são carregados do banco de dados etc.

O código compilado do aplicativo também vai para a memória. É possível alterar a área de memória onde ele está. Sim, é isso mesmo. Alterar os bits de código diretamente na memória RAM, ou seja, alterar o comportamento do aplicativo enquanto ele é executado. Modificar o código que foi lido do disco rígido. Obviamente existem restrições. Já pensou se qualquer aplicativo pudesse alterar o código de outro em execução? Seria um prato cheio para desenvolvedores de vírus.

Nota do DevMan

As extensões SSE3 (Streaming SIMD Extensions 3) permitem um alargamento das capacidades SIMD disponibilizadas pelas extensões SSE e SSE2. Permitem um aumento do desempenho no processamento de dados em diversas áreas como, por exemplo, a aritmética de números complexos e na descodificação de vídeo. Mais informações em http://pt.wikipedia.org/wiki/SSE3

Nota do DevMan

John von Neumann: é considerado um dos mais importantes matemáticos do século XX. Foi membro do Instituto de Estudos Avançados em Princeton, New Jersey, do qual também faziam parte Albert Einstein. A maioria dos computadores de hoje em dia segue ainda o modelo proposto por von Neumann. Esse modelo define um computador sequencial digital em que o processamento das informações é feito passo a passo, caracterizando um comportamento determinístico (ou seja, os mesmos dados de entrada produzem sempre a mesma resposta).

Em nosso caso o próprio aplicativo se alterará, quando iniciar sua execução. Do contrário, receberíamos um *Access Violation*. Faz mais sentido agora o que é um *Access Violation*? É uma simples tentativa de acessar ou alterar memória que não te pertence!

Resumindo, o que é feito é uma alteração nas páginas de memória que o executável ocupa, logo após ele ser carregado na memória. Desta forma, o executável que está no disco rígido será diferente do executável sendo executado na memória RAM.

Neste artigo será exemplificado como se pode fazer tais substituições, otimizando o método *FindField* da classe *TFields*. Este método é lento porque faz um laço varrendo todos os campos do *TDataSet* à procura de determinado campo. É o método chamado quando se executa *TDataSet.FieldByName('NOME')* por exemplo.

Não se engane pensando que, porque você não usa este método nos seus fontes, ele não é usado no seu aplicativo. Na verdade a própria VCL faz uso extensivo deste método, pois precisa ligar os seus próprios componentes aos *TFields* do *DataSet*. Isso inclui *TDBEdits*, *TDBGrids*, geradores de relatórios e qualquer outro componente que faz uso de qualquer classe descendente de *TDataSet*.

Analisando a complexidade do método FindField

Como dito na introdução, o algoritmo *FindField* é lento porque ele simplesmente varre todos os campos do *DataSet*, em ordem de índice, até encontrar o campo procurado. Isso é feito porque o campo é procurado através de seu nome.

Observando o código da **Listagem 1,** podemos fazer a seguinte análise de complexidade, a respeito do algoritmo *FindField*. Observe que a entrada do algoritmo é a lista de campos do *DataSet*, com *n* entradas/campos:

• No melhor caso é Ω(1). O campo procurado é o primeiro da lista. O laço *for* da linha 5 será executado apenas uma vez e o *exit* da linha 8 será então chamado.

• No caso médio é Θ $\left(\frac{1+2+...+n}{n}\right)$ se todos os campos forem consultados a cada iteração.

• No pior caso é O(n). O campo procurado é o último ou não existe. O nome do campo foi comparado com todos os outros.

Listagem 1. Código do método FindField da VCL

```
1. function TFields.FindField(const FieldName: string): TField;
2. var
3.   I: Integer;
4. begin
5.   for I := 0 to FList.Count - 1 do
6.   begin
7.     Result := FList.Items[I];
8.     if WideCompareText(Result.FFieldName, FieldName) = 0 then Exit;
9.   end;
10.  Result := nil;
11. end;
```

Existem pelos menos duas formas de se otimizar este código: Através de uma lista ordenada ou uma lista *Hash*. Com uma lista ordenada conseguiríamos um tempo O(lg n) sempre, indiferente se o campo está no começo ou final da lista. Esta abordagem introduz um *overhead* devido à necessidade de ordenação desta lista. Poderíamos realizar esta ordenação em tempo Θ (n lg n), utilizando o algoritmo *quicksort* por exemplo. A ordenação deveria ser executada uma vez a cada abertura do *DataSet*. Escrevi um artigo sobre este método no meu blog pessoal, no artigo "*FieldByName* é O(n), que tal melhorá-lo para O(Lg n)?" que está disponível em *www.thborges.com/?p=36*.

Irei um pouco além agora. Explicarei como otimizar o método *FindField* utilizando uma tabela *Hash*. Esta abordagem garante um tempo constante de praticamente O(1), indiferente da posição que o campo ocupa no *DataSet*. É como se a função sempre soubesse onde está o campo na lista, olhando apenas para seu nome.

O que é uma tabela Hash?

Uma tabela *Hash* é uma estrutura de dados que armazena um conjunto de itens em um *array*, utilizando uma função de espalhamento para determinar o índice onde cada item será armazenado. A função de espalhamento, ou função *hash*, opera sobre o conjunto de itens, convertendo cada um deles em um número inteiro. Este número inteiro é utilizado como índice no *array* onde o item será armazenado.

Diz-se que esta função é **perfeita** se a mesma consegue atribuir a cada item da lista um número único, sem repetições. E pode ainda ser **mínima** se faz este mapeamento utilizando a menor quantidade de índices possíveis, sem repetir. Observe na **Figura 1** que cada nome foi mapeado em um índice do vetor, mas sobraram posições no vetor. Esta função é dita perfeita, mas não mínima. Ao contrário, na **Figura 2**, observe que não há posições não utilizadas no vetor, portanto a função é mínima.

Nota do DevMan

Os termos melhor caso, caso médio e pior caso, representados pelos símbolos Ω, Θ e O respectivamente, vem da disciplina Complexidade de Algoritmos. Dizem respeito ao tempo que um algoritmo leva para concluir seu processamento, em relação ao tamanho da entrada que lhe é dada. Neste exemplo a entrada é a lista de todos os campos. O(n) quer dizer que todos os campos da lista (n campos) serão consultados para se chegar à resposta do processamento. Complexidades comuns na literatura são O(n lg n) e O(n²) para algoritmos de ordenação de listas, O(n³) para busca de caminhos mínimos em grafos, dentre outros. Esta disciplina é estudada geralmente em cursos na área de computação.

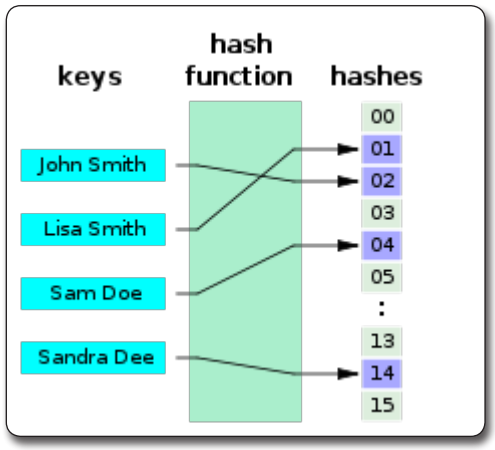


Figura 1. Função hash perfeita (fonte: Wikipedia)

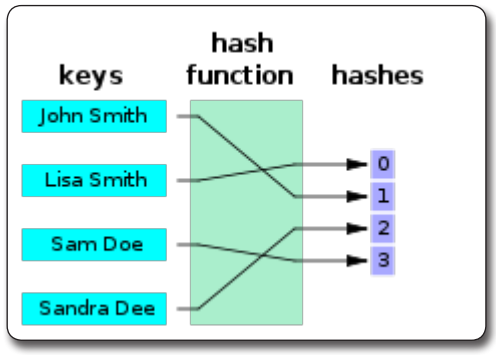


Figura 2. Função hash mínima (fonte: Wikipedia)

Na prática, funções *hash* mínimas e perfeitas são difíceis de obter. Em geral a função retorna índices repetidos para itens distintos. Por esse motivo, em cada posição do *array* é armazenado não somente um, mas vários itens. De forma que, quanto mais índices repetidos a função retornar, pior será a estrutura de dados em relação ao tempo de busca de um determinado item. Mas como uma tabela *hash* pode melhorar o tempo de busca de itens em uma lista? No nosso caso temos campos nomeados, cujos nomes não podem se repetir para uma determinada tabela. Durante o processo de inserção dos campos no *DataSet* iremos executar uma função *hash* que opera sobre *strings*, passando como parâmetro o nome do campo. A função irá retornar um número inteiro.

Digamos que para o campo de nome CODCLIENTE a função retorne 10. Incluiremos então o campo na posição 10 de um *array*. Faremos isso para todos os campos. Quando o método *FindField* for executado, pegaremos seu parâmetro (o nome do campo procurado) e chamaremos a função *hash*. Esta devolverá exatamente o índice do *array* (10) no qual o campo foi armazenado.

Há uma particularidade neste método. As formas mais simples de tabelas *hash* exigem que a quantidade de posições no *array* seja previamente conhecida. Os limites da função *hash* vão, por exemplo, de 0 a 64. Isso quer dizer que a função nunca irá retornar um número maior que 64, independente da *string* passada como parâmetro. Para os campos de um *DataSet* isso pode ser um exagero, visto que geralmente não se tem uma tabela com tantos campos. O ideal é que se identifique um valor que cause a menor quantidade de repetições de índices possível. Se a função *hash* fosse perfeita e mínima, este valor seria a quantidade de campos no *DataSet*.

Para nosso propósito utilizaremos a função *hash* codificada pelo projeto *DelphiSpeedUp*. A função *hash*, apesar de simples, foi codificada utilizando a linguagem *assembly*. O código *assembly* da função está na **Listagem 2**. Nas linhas 5 a 7 calcula-se o tamanho da *string*. A seguir nas linhas 10 a 26 existe um *loop* percorrendo os caracteres da *string*. Observe que o @@ indica um *label* e a instrução *jnz* é como um *goto* que volta para este *label*, enquanto não chegar ao fim da *string*. Na primeira etapa, linhas de 11 a 17, é feita uma operação que desconsidera letras maiúsculas e minúsculas. Observando a ordem dos caracteres na tabela ASCII, subtrair 'a' - 'A' de um caractere minúsculo, corresponde a transformá-lo em uma letra maiúscula. Nas linhas 20 a 23 é feito o cálculo do *hash*, somando o ordinal de cada caractere ao resultado. Por fim na linha 28 faz-se a operação de módulo 64, que é o resto da divisão do número encontrado por 64. Note que o resto desta divisão nunca é maior que 63 e isto garante que todos os campos serão armazenados em 64 posições de um *array*.

Passo a passo da alteração na classe TFields

A otimização do método *FindField* será feita em quatro passos, como descrito a seguir:

1. Cria-se uma nova *unit*, onde serão declaradas as classes auxiliares e executados os métodos de substituição na seção *initialization*. Chamaremos de *FieldOptimize.pas*;
2. Cria-se a classe *TNameHashList* que será a lista *hash*, descendente de *TList*. Será usada para substituir a lista de campos da classe *TFields* (veja detalhes a seguir);
3. Cria-se uma cópia parcial da classe *TFields*, para poder acessar seus campos privados através de um *typecast* forçado. Chamaremos de *TPrivateFields*;
4. Cria-se a classe *TFastFields* que conterá apenas o método *FindField* otimizado.

A classe TNameHashList

A classe *TNameHashList* é uma variação da classe de mesmo nome do projeto *FastCode*, mais precisamente da *unit* *RtlVCLOptimize.pas*. É necessária apenas uma adaptação para que ela opere com *TFields*.O ponto mais importante da construção é a reescrita do método *Notify*. Este método é invocado pela *TList* sempre que um item é adicionado ou removido da lista. Assim, sempre que isso acontecer estaremos alimentando um vetor adicional, que é o *hash* em si.

Três métodos são necessários: *NameAdd*, *NameRemove* e *NameFind*. Estes métodos serão chamados para adicionar, remover e procurar um item na lista *hash*, respectivamente.

Observe na **Listagem 3** o método *NameAdd*. Na linha 8 é chamado o método *HashupString* (descrito na **Listagem 2**) armazenando seu retorno em *AHash*. *AHash* representa o índice onde o *TField* deve ser inserido. Pode ter ocorrido uma colisão de índices e a posição *AHash* já está sendo usada por outra entrada. Salva-se então, Linha 9, o conteúdo corrente da posição *AHash* no campo *Next* de uma nova estrutura N do tipo *NameCompItem*, formando uma lista ligada (veja Nota do DevMan). A seguir, linhas 10 a 13, a posição *AHash* do *array* é substituída pela nova entrada e o par *Key/Value* da estrutura N é preenchido com o nome do campo e sua instância, respectivamente.

Nota do DevMan

Cada caractere na tabela ASCII é representado por um número. Inicialmente foram definidos 128 caracteres, de 0 a 127, e incluem caracteres de controle (quebra de linha, retorno de carro, além de outros) e caracteres que possuem representação gráfica (as letras, números e símbolos). Quando se subtrai uma letra por outra, obtém-se um número. No caso da função HashUpString, subtrair 'A' – 'a' é o mesmo que subtrair 97 – 65 = 32. Desta forma, subtrair 32 de uma letra minúscula qualquer a transforma em uma letra maiúscula. Por exemplo, a letra 't' (116) – 32 = 'T' (84).

Listagem 2. Função HashupString do projeto DelphiSpeedUp

```
1.  function HashUpString(const AItem: string): Integer;
2.  asm
3.   or eax, eax
4.   jz @@Leave
5.
6.   xchg eax, edx
7.   mov eax, [edx-$04] // Length(AItem)
8.   xor ecx, ecx
9.
10. @@HashStringNextChar:
11.  mov cl, [edx]
12.
13.  cmp cl, 'a'
14.  jb  @@UpCaseEnd
15.  cmp cl, 'z'
16.  ja  @@UpCaseEnd
17.  sub cl, 'a' - 'A'
18. @@UpCaseEnd:
19.
20.  ror cl, 4
21.  shl cx, 1
22.  add eax, ecx
23.  xor ch, ch
24.  inc edx
25.  or ecx, ecx
26.  jnz @@HashStringNextChar
27.
28.  and eax, 64-1
29. @@Leave:
30. end;
```

Nota do DevMan

Observe que a lista ligada é necessária devido a possibilidade de colisões, conforme pode ser observado na **Figura 3**. Veja que Sandra Dee e John Smith estão posicionados no índice 152 do array. É neste detalhe que está a velocidade ou lentidão de uma lista hash. Se esta lista ligada ficar muito longa, ou seja, ocorrerem muitas colisões, o processo de busca ficará lento.

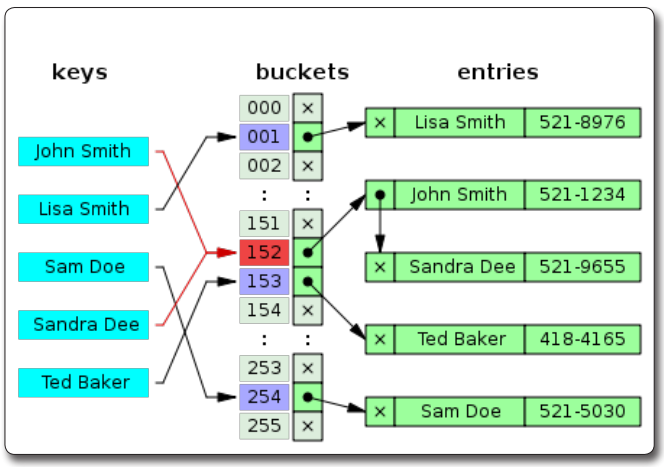


Figura 3. Tabela Hash com colisão na posição 152. Observe que John Smith está na mesma posição que Sandra Dee (fonte: Wikipedia)

Listagem 3. Função NameAdd da classe TNameHashList

```
1.  function TNameHashList.NameAdd(const AItem: string;
2.    AData: TField): TField;
3.  var
4.    N: PNameCompItem;
5.    AHash: Integer;
6.  begin
7.    New(N);
8.    AHash := HashUpString(AItem);
9.    N.Next := FItems[AHash];
10.   FItems[AHash] := N;
11.   Inc(FNameCount);
12.   N.Key := AItem;
13.   N.Value := AData;
14.   Result := AData;
15. end;
```

O que a função *NameRemove* faz é remover um item da lista *hash*. Inicialmente chama-se *HashupString* com o nome do campo que será removido. Como na inserção também foi utilizada esta função para escolher em que índice o campo seria inserido, o mesmo só pode estar na lista ligada da posição retornada, digamos que seja a posição P. Procura-se então na lista ligada do índice P pelo respectivo campo. O campo pode estar no início, meio ou fim da lista, portanto ao remover é necessário manter os links entre os itens corretamente, cada um apontando sempre para o seguinte. O código foi omitido para evitar poluição do texto, mas está disponível no download do artigo.

Por fim, a função *NameFind*, **Listagem 4**, é responsável por encontrar um campo na lista *hash*. Encontra-se a posição *AHash* no *array*, onde o campo pode ter sido armazenado, utilizando o nome do campo procurado, Linha 8. A lista ligada desta posição é então percorrida, Linhas 10 a 19, até que se encontre um campo com o nome procurado. Caso o campo não exista, a lista será percorrida item a item.

Observe que existe uma situação de pior caso, onde todos os campos estão armazenados na mesma posição do *array*, tornando a lista muito grande. A procura neste caso seria tão ruim quanto a do método original. Porém, no caso de *TFields*, esta lista tende a não ser muito grande devido os nomes dos campos serem sempre diferentes uns dos outros. Fica evidente aqui também a importância de escolher um tamanho adequado para o *array* da lista *hash*.

As classes TPrivateFields e TFastFields

Fazendo uma analogia, será necessário fazer uma "cirurgia" na classe *TFields*, em tempo de execução. Fazer um transplante de sua lista de campos *FList*, digamos o seu coração, e conectar algumas artérias (método *FindField*) a outros locais. Porém, o campo *FList* da classe *TFields* é privado. Precisaremos utilizar um *typecast* um pouco forçado para conseguir acessá-lo, destruí-lo, e substituir por uma instância de *TNameHashList*.

A ciência está por trás do *typecast*. Ensinaram-te que o *typecast* "transforma" uma classe em outra, certo? Mas além do básico, o que está acontecendo é apenas uma manipulação de endereços de memória. É você dizendo ao compilador: *"Acesse os campos deste objeto como se ele fosse este outro"*. E então o compilador começa a gerar o executável escrevendo: O primeiro campo do objeto está na posição onde o objeto está instanciado + 4 bytes, o segundo na posição inicial + 8 bytes e por aí vai. Tudo se resume a endereços de memória e intervalos.

Declara-se então uma nova classe *TPrivateFields*, idêntica ao início de *TFields*. Descendendo da mesma classe (*TObject*) que a *TFields*, declarando os campos da seção *private* até chegar no desejado. Todos os campos declarados têm que estar na mesma ordem e posição.

Desta forma, como todos os campos estão na mesma ordem, e portanto com os mesmos intervalos de endereços de memória do objeto original, podemos acessar qualquer campo privado do objeto como se fosse o nosso próprio.

Listagem 4. Declaração parcial da classe TFields na unit DB.pas

```
1. function TNameHashList.NameFind(const AItem: string;
2.   out Value: TField): Boolean;
3. var
4.   N: PNameCompItem;
5.   AHash: Integer;
6. begin
7.   Value := nil;
8.   AHash := HashUpString(AItem);
9.   N := FItems[AHash];
10.   while N <> nil do
11.   begin
12.     if CompareText(N.Key, AItem) = 0 then
13.     begin
14.       Value := N.Value;
15.       Result := True;
16.       Exit;
17.     end;
18.     N := N.Next;
19.   end;
20.   Result := False;
21. end;
```

Mas atenção, os desenvolvedores da VCL podem mudar esta ordem de uma versão para outra, fique atento.

Na **Listagem 5** está a declaração da classe *TPrivateFields*. Note que o *FList* é o primeiro campo da seção *private*, e por isso ela ficou bem simples. O segredo maior está na classe *TFastFields*. Ela existe apenas para conter a função *FindField*. Sequer será instanciada durante toda a execução do projeto. Nela declaramos *FindField* com a mesma assinatura da *TFields.FindField*, Linhas 8 a 11. A implementação deste método consiste em testar se a *FList* já é uma instância de *TNameHashList* e, caso contrário, substituí-la. Para isso, faz-se o *typecast* para acessar o campo privado *FList,* Linha 20. Caso não seja (Linhas 22 a 28), instancia-se uma nova *TNameHashList* (Linha 22) copia-se os campos porventura já existentes na lista original (Linhas 24 a 26), substitui-se a lista antiga pela nova *TNameHashList*, Linha 27) e por fim destrói-se a lista antiga, Linha 28. Mas atenção, essa substituição só funciona porque a *TNameHashList* também é descendente de *TList*. Note que todo o código da classe *TFields* continua acessando o campo privado *FList* como se nada tivesse acontecido.

Não ocorre vazamento de memória porque a própria classe *TFields* se encarregará de destruir a tabela hash. Na verdade ela irá destruir o conteúdo da variável *FList*, achando que lá existe uma *TList* comum, mas sabemos que isso não é verdade porque a substituímos. Porém como o método *Free* é herdado de *TObject*, não haverá nenhum problema.

A mágica da substituição de código com JMP ($E9)

Na memória RAM do computador há uma quantidade enorme de números que possuem os mais diversos significados. Existem números que representam dados, como letras e números, e números que representam operações que o computador deve fazer. É o chamado conjunto de instruções do computador, como por exemplo *SSE2* ou *SSE3*.

Há um conjunto padrão destas instruções, válidos para a maioria dos processadores. Estas operações incluem soma, subtração, redirecionamentos e muitos outros. O que nos importa aqui é o redirecionamento, ou se preferir, o *goto.* Ele é representado pelo número hexadecimal *$E9*, instrução *assembly JMP*, que significa fazer um salto, *jump*, para um endereço de memória e continuar o processamento deste endereço em diante. Note que existem vários tipos de *jumps*. Há *jumps* que só saltam se determinada condição for atendida, se o número for maior, igual, menor que outro, por exemplo. Este que usaremos é o incondicional.

A substituição de uma função por outra é feita escrevendo no início da função *FindField* original um *goto* para o endereço da nova função. A função original é referenciada em muitos locais na memória, mas sempre que o processador chegar no endereço dela será forçado a ir para outro endereço: o endereço da nova função.

Observe na **Listagem 6** a função *CodeRedirect*, extraída do projeto *DelphiSpeedUp*. A função recebe o endereço de dois métodos como parâmetros, *Proc* e *NewProc*. Primeiro captura-se o endereço na memória onde o procedimento original está.

Listagem 5. Código das classes TPrivateFields e TFastFields

```
1. type
2.
3.   TPrivateFields = class(TObject)
4.   private
5.     FList: TList;
6.   end;
7.
8.   TFasTFields = class(TFields)
9.   public
10.    function FindField(const FieldName: WideString): TField;
11.  end;
12.
13. { TFastFields }
14.
15. function TFastFields.FindField(const FieldName: WideString):
     TField;
16. var
17.   List, HashList: TList;
18.   I: Integer;
19. begin
20.   if not (TPrivateFields(Self).FList is TNameHashList) then
21.   begin
22.     HashList := TNameHashList.Create;
23.     List := TPrivateFields(Self).FList;
24.     HashList.Capacity := List.Capacity;
25.     for I := 0 to List.Count - 1 do
26.       HashList.Add(List.List[I]); // copy and hash
27.     TPrivateFields(Self).FList := HashList;
28.     List.Free;
29.   end;
30.
31.   TNameHashList(TPrivateFields(Self).FList).
       NameFind(FieldName, Result);
32. end;
```

Para isso utiliza-se a função *GetActualAddr*, Linha 7. A seguir, linhas 8 e 9, é solicitado ao sistema operacional que permita a escrita na página onde a função está localizada (Função *VirtualProtectEx*, com parâmetro *PAGE_EXECUTE_READWRITE*). Na linha 11 e 12 é então feito o redirecionamento, utilizando o *record TInjectRec*. Este *record* serve para injetar o código *assembly* de salto. No primeiro byte grava-se a operação (*jump*) e nos quatro bytes seguintes a quantidade de bytes a saltar, a partir do endereço atual. Por fim volta-se a página de memória para o estado anterior, provavelmente *read-only*, e executa-se um *flush* para descarregar a página dos registradores, evitando problemas com máquinas de vários núcleos, linhas 14 a 16.

Listagem 6. Função CodeRedirect do projeto DelphiSpeedUp

```
 1. procedure CodeRedirect(Proc: Pointer; NewProc: Pointer);
 2. var
 3.   OldProtect: Cardinal;
 4. begin
 5.   if Proc = nil then
 6.     Exit;
 7.   Proc := GetActualAddr(Proc);
 8.   if VirtualProtectEx(GetCurrentProcess, Proc, SizeOf(TInjectRec),
 9.     PAGE_EXECUTE_READWRITE, OldProtect) then
10.   begin
11.     TInjectRec(Proc^).Jump := $E9;
12.     TInjectRec(Proc^).Offset := Integer(NewProc) - (Integer(Proc) +
13.       SizeOf(TInjectRec));
14.     VirtualProtectEx(GetCurrentProcess, Proc, SizeOf(TInjectRec),
15.       OldProtect, @OldProtect);
16.     FlushInstructionCache(GetCurrentProcess, Proc,
         SizeOf(TInjectRec));
17.   end;
18. end;
```

Colocando a cereja no bolo

Por fim e o mais importante de tudo: o redirecionamento da função *FindField* da classe *TFields* para a classe *TFastFields*. Observe a **Listagem 7**. Basta fazer uma chamada à função *CodeRedirect*, passando como parâmetro o endereço do método origem e destino.

Listagem 7. Redirecionamento de TFields.FindField para TFastFields.FindField

```
1. initialization
2.   CodeRedirect(@TFields.FindField, @TFastFields.FindField);
```

O que acontecerá quando o aplicativo iniciar?

1. O código da seção *initialization* será invocado. Ele alterará a página de memória onde foi colocado o método *TFields.FindField*, forçando o processador a saltar para o endereço onde o método *TFastFields.FindField* foi escrito na memória, sempre que a primeira função for chamada;
2. A função *TFields.FindField* é chamada em qualquer local do aplicativo e o processador salta para a *TFastFields.FindFind*;
3. A nova função *FindField* testa se a *FList* ainda é uma instância de *TList*. Como é a primeira chamada ela trocará a *TList* por uma *TNameHashList*;
4. Os campos são adicionados na lista *hash* à medida que vão sendo inseridos, através do método *Notify* reescrito;
5. O campo é posteriormente procurado na *TNameHashList* e retornado como se o nosso transplante não tivesse acontecido. Porém, agora a complexidade da busca é O(1), desconsiderando as colisões no *hash*.

Conclusão

Este tipo de intervenção em tempo de execução é uma forma avançada de corrigir ou melhorar códigos de bibliotecas, muito utilizado pelos projetos *FastCode* e na maioria das melhorias feitas por *Andreas Hausladen* (andy.jgknet.de/blog), dentre elas o projeto *DelphiSpeedUp*.

Uma das melhorias introduzidas por tal projeto é exatamente colocar uma lista *hash* para ser utilizada nas chamadas do método *FindComponent*. Boa parte da otimização é essa. O *FindComponent* da *VCL* é feito da mesma forma que o *FindField*. Existe uma lista de componentes que é preenchida sempre que objetos descendentes de *TComponent* são criados. Ao procurar por um componente esta lista é varrida item a item, comparando os nomes dos componentes até encontrar o desejado.

A melhora com a otimização da função *FindField* pode ser percebida principalmente no tempo de carregamento das telas, onde acontece a maioria das chamadas a esta função, vinculando os *TDBEdits* e correlatos aos campos nos *DataSets*. Nos *DataModules*, quando se usa campos persistidos, ocorre uma vinculação entre todos os *TFields* logo em sua abertura. O mesmo acontece quando se abre os *TDataSets*. Outra situação onde ocorre melhora é quando se usa o método *FieldByName* em rotinas que processam grandes quantidades de registros.

O uso de listas *hash* pode ser expandido a qualquer situação onde seja necessário localizar objetos através de seus nomes. O único detalhe a observar é o tamanho do *array* do *hash*, onde serão colocados os itens. É necessário ter uma ideia da quantidade de itens que serão utilizados. É importante considerar também o quão bem a função *hash* espalha os itens nas posições deste *array*. Dependendo da rotina, uma boa ideia é deixar esta configuração como parâmetro do sistema.

Para quem utiliza versões mais atuais do *Delphi*, outro ponto possível é a utilização de *generics* para construção de uma lista *hash* que possa ser utilizada com qualquer objeto que possua um campo representando seu nome.

Outras otimizações na VCL ou em componentes de terceiros podem ser feitas. Porém, encontrá-las e saber como otimizar é uma tarefa um pouco mais complicada. Depende da experiência e conhecimento de cada pessoa. Mas a forma de corrigir sem alterar fontes é esta. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

A partir de agora você pode contar com a ajuda da

Chegou a Consultoria On-line DevMedia

Consultoria Técnica + Professor Virtual + Certificação

Mais Informações:
www.devmedia.com.br/consultoria_online (21) 3382-5038

DevMedia em seus projetos e estudos.

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web a Consultoria DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance uma consultoria de qualidade por um preço muito acessível. Consulte nossos planos.

Mais um serviço DevMedia group

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Curso de Delphi Prism – Parte 3

Novidades da Linguagem – Classes parciais, Tipos anônimos e Métodos de extensão

Guinther Pauli
guinther.pauli@gmail.com
Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software, dos quais 14 dedicados ao Delphi e 9 ao .NET/C#. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Delphi para Linux, Certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET, VB.NET, MSF, SQL Server). Editor geral da Revista ClubeDelphi, Editor Geral da Revista .NET Magazine, professor de Pós-Graduação. Publicou mais de 150 artigos, 400 vídeo-aulas, palestrante em 5 edições da Borland Conference. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adotação de melhores práticas. Blog: guintherpauli.blogspot.com

Resumo do DevMan

Neste artigo falaremos essencialmente de classes e como elas se comportam no Delphi Prism. O recurso de classes parciais permite que o código de uma mesma classe possa ser separado em diferentes arquivos de códigos-fontes. Tipos anônimos podem ser usados para, em uma mesma linha, declarar, instanciar e inicializar um objeto sem tipo. E classes / métodos de extensão permitem injetar código em classes pré-existentes, sem usar herança ou mesmo sem ter acesso ao código-fonte da classe alvo.

Do que trata o artigo
Lançado pela Embarcadero, empresa que adquiriu a CodeGear, o Delphi Prism é a nova forma de desenvolver aplicações para o .NET Framework e Mono utilizando o Object Pascal e o IDE do Visual Studio 2008. Nesta terceira parte do curso, apresento mais sobre os novos recursos da linguagem utilizada pelo Delphi Prism, que são inúmeros se compararmos com o Delphi for .NET tradicional ou mesmo Delphi Win32

Para que serve
Comparar a sintaxe do antigo Pascal e Delphi tradicional com a moderna e poderosa linguagem suportada pelo Delphi Prism, conhecer os principais novos recursos da linguagem e vantagens

Em que situação o tema é útil
O Delphi Prism abre uma porta incrível para os desenvolvedores Delphi, pois podem agora utilizar um IDE robusto, rápido e estável para construir aplicações para .NET, incluindo as novas tecnologias ASP.NET 3.5, ASP.NET AJAX, WPF, WCF, MONO etc. usando uma versão evoluída do Object Pascal, que mescla os melhores recursos do Delphi e do C# 3.0

As classes fazem parte da vida do desenvolvedor *Delphi* desde a sua primeira versão. Mas as coisas não eram simples no tempo do *Pascal*. Para quem lembra, tínhamos que fazer algo como na **Listagem 1.** Este fragmento de código foi retirado de um projeto que desenvolvi ainda nos tempos acadêmicos (1996), disponível com os fontes em no meu site no *CodeCentral* da *Embarcadero* (veja seção *Links*). Dados e funcionalidade ficavam separados, fazíamos um *record* para guardar os dados de uma entidade, depois uma variável *file* para gravar o *record* no disco e um *buffer* de memória para manipular o registro. E separadamente, os *procedures* para manipular esses dados.

Listagem 1. Records, variáveis globais e programação estruturada

```
TCliente = record
    codigo: integer;
    ai: char; // undelete à moda antiga
     nome: string39;
  ...
var
Fclientes : file of TCliente;
Bclientes : TCliente;
...
Procedure open_Fclientes;
Procedure close_Fclientes;
Procedure Cria_Fclientes;
Procedure zera_cli;
Procedure menu_cli;
Procedure mostra_cli;
```

O suporte a objetos veio no *Turbo Pascal 5.5*, com a inclusão da palavra-chave *object*. Basicamente podíamos colocar os *procedures* que trabalhavam com os dados do *record* dentro do próprio *record* (que então virou *object*). Mais tarde, no *Delphi 2*, a palavra-chave foi renomeada para *class*, por questões óbvias, convenção que permanece até hoje, inclusive no *Prism* e *C#*. O *record* ainda existe, e evoluiu, podemos no *Prism* colocar *procedures* dentro de *records*. E em algumas versões do *Delphi for .NET* também. Ele é usado ainda hoje quando queremos criar estruturas que são passadas por valor, e não por referência, como o que acontece no caso de classes.

Tudo bem, mas o que isso tem a ver com esse curso? Novamente, estamos olhando para o passado para entender o presente e futuro. Vamos ver como seria minha antiga classe *TCliente* no *Delphi Win32* mais próximo da atualidade (**Listagem 2)**. Note alguns pontos importantes: a classe *TCliente* possui duas propriedades, *Nome* e *Idade*. Tudo bem, seria mais interessante guardar a data de nascimento, mas por questões didáticas preferi usar um tipo *integer* para simplificar o exemplo. As propriedades são amparadas por métodos *get / set*, que reforçam o princípio de encapsulamento no *Object Pascal*. Esses métodos são privados e nada mais fazem do que mapear os dados das propriedades públicas para campos privados. Na **Listagem 3** temos um exemplo de uso da classe.

E no *Delphi Prism*? Podemos fazer exatamente da mesma forma, igualzinho, o código compila sem problema algum. Porém, alguns novos recursos da linguagem, muitos oriundos do *C# 2.0* e *3.0*, permitem fazer a mesma coisa

Listagem 2. Classes no Delphi Win32

```
unit uCliente;

interface

type
  TCliente = class
  private
    FIdade: integer;
    FNome: string;
    function GetNome: string;
    procedure SetNome(const Value: string);
    function GetIdade(): integer;
    procedure SetIdade(const Value: integer);
  public
    property Nome: string read GetNome write SetNome;
    property Idade: integer read GetIdade write SetIdade;
  end;

implementation

{ TCliente }

function TCliente.GetIdade(): integer;
begin
  result := FIdade;
end;

function TCliente.GetNome(): string;
begin
  result := FNome;
end;

procedure TCliente.SetIdade(const Value: integer);
begin
  // Alguma validação
  FIdade := Value;
end;

procedure TCliente.SetNome(const Value: string);
begin
  // Alguma validação
  FNome := Value;
end;

end.
```

Listagem 3. Instanciando uma classe no Delphi Win32

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Cliente: TCliente;
begin
  Cliente := TCliente.Create();
  Cliente.Nome := 'Ben Campbell';
  Cliente.Idade := 21;
end;
```

com código extremamente reduzido. Veja na **Listagem 4** o mesmo exemplo anterior do *Delphi Win32*, agora adaptado para o *.NET* com *Prism* (como comentei no começo do curso, usaremos sempre aplicações *ASP.NET – File>New Web Site*). Aqui criei uma nova classe, no arquivo *Cliente.pas* (pasta *App_Code*), para isso, basta dar um duplo clique no nome do projeto no *Solution Explorer* e escolher *Add New Item>Class*. A classe se chama *TCliente* e tem duas propriedades. Note que, se não formos incluir alguma validação ou algum tratamento personalizado nos métodos *get/set*, podemos omiti-los. Também não precisamos declarar o campo privado. Esse recurso é conhecido como propriedades automáticas. A propósito, o C# tem o mesmo recurso, como vemos o código equivalente na **Listagem 5**. Esse tipo de classe é muito útil para criar *DTO's* (*Data Transfer Objects*), para trafegar dados entre camadas em uma aplicação *.NET* (veja meu artigo da edição 98). Também usado em aplicações com *frameworks* de persistência e mapeamento objeto / relacional.

Listagem 4. Classes no Delphi Prism, arquivo Cliente.pas

```
TCliente = public class
public
  property Nome: String;
  property Idade: Integer;
end;
```

Listagem 5. Classes no C#

```
public class TCliente
{
    public string Nome { get; set; }
    public string Idade { get; set; }
}
```

Nota

Ao longo deste curso, vou fazer várias referências ao C#. Mesmo usando o Delphi Prism, é importantíssimo que você conheça a semântica da linguagem C#, pois ao desenvolver para .NET, toda documentação, livro, artigo e outros recursos que você for utilizar, provavelmente será escrito em C# ou VB.NET.

Para usar a classe, podemos fazer algo como na **Listagem 6.** Para ilustrar o uso da classe, coloquei um *GridView* no *Web Form*. Criei três clientes, adicionei-os em uma coleção genérica (usando *generics*) e atribui essa lista à variável *DataSource* do *GridView*. O resultado é visto na **Figura 1**. Para fazer acesso à classe que está no arquivo *cliente.pas*, não precisamos colocar nada no *uses* do arquivo do *Web Form*. Isso porque aqui não existe o conceito de *unit* e sim *namespace*. Se não coloquei a classe em um *namespace* especial, então ele está no mesmo *namespace* default do projeto, logo, não precisa importar (que belo recurso!). Para instanciar clientes, usamos variáveis *in-line*, discutidas nas partes anteriores do curso (basicamente podemos agora declarar, instanciar e inicializar uma variável dentro do código, como no C#). Para instanciar objetos, usamos o novo operador *new* e não *create*. Usamos também construtores estendidos (também conhecido como *object initializer*), recurso que permite que passemos os valores das propriedades da classe sendo instanciada no parâmetro do construtor, sem a necessidade de criar *overloads* ou código extra após a criação do objeto (tipo o que fazíamos com *with create do*).

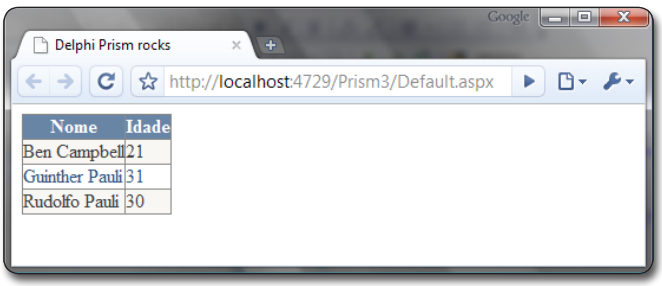


Figura 1. Objetos sendo exibidos no GridView

Agora vamos fazer um teste. Primeiro vamos gerar uma versão de *build* da aplicação *ASP.NET*, acessando o menu *Build > Publish Web Site*. Isso vai compilar o *.pas* e o *.aspx* (caso tenha desmarcado a opção para não gerar *aspx*) em um arquivo DLL, que é um *assembly .NET*. Depois, abrimos o *Visual Studio 2008 Command Prompt* na

Listagem 6. Usando a classe

```
namespace;

interface

uses
  System,
  System.Web,
  System.Web.UI,
  System.Web.UI.WebControls,
  System.Collections.Generic;

type
  _Default = public partial class(System.Web.UI.Page)
  protected
    method Page_Load(sender: Object; e: EventArgs);
  end;

implementation

method _Default.Page_Load(sender: Object; e: EventArgs);
begin
  var Ben := new TCliente(Nome := "Ben Campbell", Idade := 21);
  var Gui := new TCliente(Nome := "Guinther Pauli", Idade := 31);
  var Rud := new TCliente(Nome := "Rudolfo Pauli", Idade := 30);
  var Clientes := new List<TCliente>();
  Clientes.Add(Ben);
  Clientes.Add(Gui);
  Clientes.Add(Rud);
  GridView1.DataSource := Clientes;
  GridView1.DataBind();
end;

end.
```

pasta do grupo de programas do *VS* no menu *Iniciar*. Digite então ILDASM (utilitário para fazer examinar o código IL descompilado da DLL) e abra a *App_Code.dll*, localizada no diretório informado no momento do *Publish*. Observe (**Figura 2**) que, mesmo usando propriedades automáticas, o *Oxygene* (o compilador do *Prism*) gerou nos bastidores as variáveis privadas e os métodos *get* e *set*. Você pode dar um *dump* para o disco do seu código descompilado em IL, usando o menu *File*.

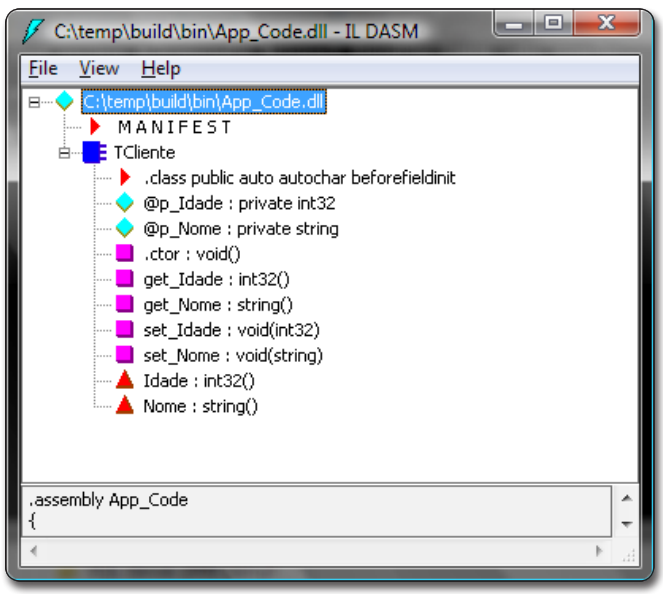


Figura 2. Examinando o código IL – Compilador gerou classe completinha

Classes Parciais

Esse recurso surgiu no *.NET Framework 2.0 / C# 2.0* e também é suportado pela versão atual do *Prism*. Em poucas palavras e indo direto ao ponto: o recurso de classes parciais permite

que você codifique parte de uma classe em um arquivo, parte em outro, todas as classes tendo o mesmo nome. No momento da **compilação**, o compilador une todos os códigos e gera uma única a classe. Mas por que motivos eu precisaria disso? Veremos a seguir.

Em nosso exemplo, para ilustrar o recurso, dividi a classe *Cliente* em dois arquivos, *cliente1.pas* e *cliente2.pas*, como podemos ver na **Listagem 7.** Para ser possível usar o recurso, usamos a palavra-reservada *partial class*. Não se preocupe, você não receberá um *Identifier redeclared*. Se executar a aplicação, tudo funcionará normalmente, a **Listagem 6** não precisa ser modificada.

Listagem 7. Classes parciais

```
[Arquivo Cliente1.pas]
TCliente = public partial class
public
  property Nome: String;
end;
[Arquivo Cliente2.pas]
TCliente = public partial class
public
  property Idade: Integer;
end;
```

Vantagens? O próprio *.NET Framework*, desde a versão 2, faz uso efetivo do recurso. Por exemplo, o código gerado para inicializar componentes do *designer* foi separado em aplicações *Windows Forms*, ou seja, duas classes, mesmo nome, mas você pode acessar os atributos (componentes) normalmente. No ASP.NET, algo similar, os controles não precisam ser mais declarados na classe de *code-behind*, devido ao uso de classes parciais nos bastidores (para comprovar, veja no código do nosso *Web Form* que o *GridView* não tem declaração, mas pode ser acessado via código, como se fosse um atributo normal da classe). Outro exemplo, o ASP.NET usa classes parciais para injetar propriedades no seu *Web Form*, com o recurso de *Profiles*, através da propriedade *Profile*.

Podemos ainda usar classes parciais para injetar código. É possível estender uma classe, desde que seja parcial e esteja no mesmo *assembly*, sem usar herança, *class helpers* ou extensão de métodos (visto a seguir). Isso dá uma maior flexibilidade à aplicação, facilitando testes, depuração, separação de lógica / interfaces etc.

Tipos anônimos

Agora vamos forçar um pouco e lembrar do desafio que eu propus na primeira parte desta série. O recurso de tipos anônimos surgiu originalmente no *C# 3.0*, disponível agora para *Delphi Prism*. Este recurso está intimamente relacionado com *object initializers* mostrado anteriormente. Um tipo anônimo é basicamente um bloco de código onde usamos um objeto, de uma classe, que não tem tipo, tudo é feito *in-line*. Estranho? Então veja na **Listagem 8** o mesmo código da **Listagem 6** adaptado para usar esse recurso.

Observe atentamente a **Listagem 8.** Onde está a classe *TCliente?* Não estamos usando-a, aliás, você pode (e deve) remover o código que define a classe do projeto. O *new* não está instanciando um *TCliente*, estamos instanciando um tipo (*class*), sem nome.

Nota do DevMan

O recurso Publish Web Site surgiu no Visual Studio 2005 (ASP.NET 2.0) e permite pré-compilar um site inteiro antes do seu deploy. Ele usa, nos bastidores, um utilitário de linha de comando chamado aspnet_compiler.exe.

Nota do DevMan

IL (Intermediate Language) é um código intermediário gerado a partir da compilação de um código-fonte .NET, seja ele C#, Delphi Prism, VB.NET etc. Ou seja, não importa em qual linguagem você programar, o compilador vai gerar um IL que será praticamente o mesmo código gerado pelo compilador de outra linguagem. O IL permite assim que diferentes linguagens possam usar tipos, herança, polimorfismo entre si como se todas fossem escritas na mesma linguagem. Isso é possível graças a especificações definidas na CLS (Common Language Specification). Veja a sessão links.

Nota do DevMan

ILDASM é um descompilador? Sim, porém ele não gera o código na linguagem original, e sim em IL. Mesmo assim, a IL é uma linguagem praticamente de alto nível, é possível examinar facilmente o código e lógica de uma aplicação compilada. Para evitar engenharia reversa do seu código e proteger a propriedade intelectual, use o Dotfuscator que acompanha o VS. Esse utilitário basicamente faz refactoring no assembly, renomeando métodos, variáveis (coloca nomes como "a","b","c"), sem no entanto alterar a funcionalidade.

Nota do DevMan

Mesmo que não tivéssemos usado classes parciais, poderíamos ter classes com o mesmo nome, desde que estejam em namespaces diferentes. Por exemplo, temos um Button do ASP.NET e um Button no Windows Forms, classes com mesmo nome mas diferentes implementações.

Nota do DevMan

Profiles é um recurso incluído no ASP.NET 2.0 (VS 2005) que tem relação com o gerenciamento de estado. Definimos uma estrutura de dados no Web.Config, como por exemplo, dados relativos ao usuário, e depois gravamos / lemos esses dados via código. As informações persistem entre diferentes sessões do usuário. Antes, para obter o mesmo efeito, era necessário gravar um cookie na máquina do usuário, gravar um ID e usar essa informação para recuperar informações do BD (como preferências) toda vez que o usuário entra em um Web Site. Profiles reduzem drasticamente o trabalho necessário para realizar essa tarefa.

Listagem 8. Anonymous Types em Delphi Prism

```
method _Default.Page_Load(sender: Object; e: EventArgs);
begin
  // não existe a classe TCliente!
  var Ben := new class(Nome := "Ben Campbell", Idade := 21);
  var Gui := new class(Nome := "Guinther Pauli", Idade := 31);
  var Rud := new class(Nome := "Rudolfo Pauli", Idade := 30);
  var Clientes := new List<Object>();
  Clientes.Add(Ben);
  Clientes.Add(Gui);
  Clientes.Add(Rud);
  GridView1.DataSource := Clientes;
  GridView1.DataBind();
end;
```

Mas e *Nome* e *Idade*? Nos bastidores, o compilador vai batizar o tipo no momento da compilação com um nome qualquer, nesse caso, *f__AnonymousType1*. Ele detecta também, pelos parâmetros passados ao construtor (que não são obviamente declarados), que o tipo anônimo tem duas propriedades, chamadas respectivamente de *Nome* e *Idade*. E por tabela, gera os atributos privados e os métodos *get / set*. Duvida? Então use o ILDASM para comprovar e veja a estrutura da classe gerada nos bastidores (**Figura 3**). Veja um detalhe interessante: nós criamos três objetos, cada um distinto do outro, por que não foram criados três tipos anônimos? O *Oxygene* detectou que os três tipos têm os mesmos atributos com os mesmos nomes e tipos (*string* e *integer*), logo, usa o **mesmo** tipo anônimo para os três. E mais, se você não remover a classe *TCliente* do projeto, verá que o *Oxygene* vai detectar que o tipo que estamos instanciando é um *TCliente* (é praticamente uma mágica!), e não cria uma classe anônima, ele detecta automaticamente pelos atributos que estamos instanciando um tipo já definido. Por esse motivo, para obter o resultado da **Figura 3**, você precisa apagar ou excluir do projeto (remover mas não apagar do disco) os arquivos que definem a classe *TCliente*. Se executar a aplicação, o resultado que obterá é o mesmo resultado da **Figura 1**. A propósito, o código equivalente em C# pode ser visto na **Litagem 9.**

Uma outra questão interessante: como nós vamos criar uma lista genérica (*List<T>*), que recebe um parâmetro *Type*, se não sabemos o tipo que estamos usando? Realmente complicado. Uma solução simples e mais correta é usar uma coleção não genérica. Ou simplesmente criar a *List* usando o tipo *Object*, já que, mesmo sendo anônima, nossa classe vai ter que herdar de alguma forma de *Object* (classe base de todos os objetos no

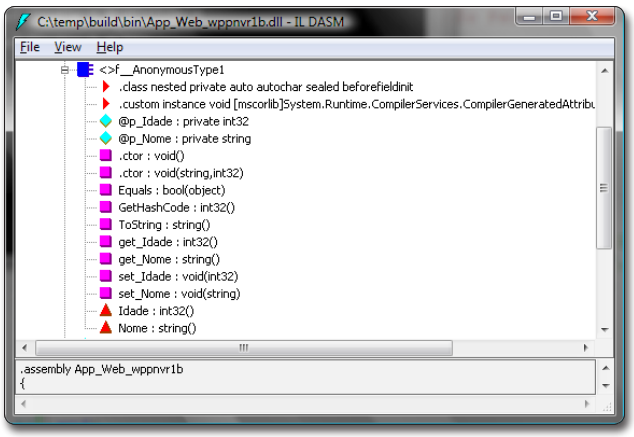


Figura 3. Tipo anônimo é batizado na compilação

Listagem 9. Anonymous Types em C#

```
protected void Page_Load(object sender, EventArgs e)
{
    var Ben = new {Nome = "Ben Campbell", Idade = 21};
    var Gui = new {Nome = "Guinther Pauli", Idade = 31};
    var Rud = new {Nome = "Rudolfo Pauli", Idade = 30};
    var Clientes = new List<Object>();
    Clientes.Add(Ben);
    Clientes.Add(Gui);
    Clientes.Add(Rud);
    GridView1.DataSource = Clientes;
    GridView1.DataBind();
}
```

.NET, equivalente ao *TObject* da VCL). Eu sei que com isso o *Generics* perde totalmente o sentido, mas pelo menos, foi preciso modificar apenas uma palavra em uma linha de código. Existem outras técnicas mais interessantes, como *factories* de tipos anônimos (veja seção *Links*). E finalmente, é perfeitamente possível usar tipos anônimos sem uma variável, por exemplo, adicionar o objeto sem tipo diretamente na *collection*, como:

```
Clientes.Add(new class(Nome := "Ben Campbell", Idade := 21));
```

Métodos de Extensão

Suponha que você queira implementar uma nova funcionalidade, por exemplo, a possibilidade de exportar os dados *TClientDataSets* para diferentes formatos além do *XML* e binário, como *HTML*, *Excel*, *TXT*, *Word* etc. (veja na seção *Links* como fazer isso). Usando boas práticas de orientação a objetos, poderíamos criar uma classe descendente de *TClientDataSet*, que implementa as novas funcionalidades, como no exemplo da **Listagem 10.** De cara já temos um problema, para usar a implementação em aplicações existentes, teremos que substituir todos os *ClientDataSets* por *TClientDataSetEx*. Outro problema: e se quisermos usar os métodos para outros tipos de *TDataSets*, como *TSqlDataSet*? Uma outra opção é criar uma *util.pas* e usar programação estruturada, parametrizando rotinas para receber um *TDataSet* (**Listagem 11**). Ou ainda, talvez a melhorzinha, criar uma classe (ex.: *TExportDataSet*) que tem uma relação de associação com um tipo *TDataSet* e possui os métodos.

Listagem 10. TClientDataSetEx

```
uses DBClient;
type
  TClientDataSetEx = class(TCustomClientDataSet)
  public
    procedure ExpHTML(FileName: string);
    procedure ExpTXT(FileName: string);
    procedure ExpXLS(FileName: string);
    procedure ExpDOC(FileName: string);
    procedure ExpXML(FileName: string);
  end;
…
```

Listagem 11. Util.pas

```
unit util.pas;

interface

uses DB;

procedure ExpHTML(DataSet: TDataSet; FileName: string);
procedure ExpTXT(DataSet: TDataSet; FileName: string);
procedure ExpXLS(DataSet: TDataSet; FileName: string);
procedure ExpDOC(DataSet: TDataSet; FileName: string);
procedure ExpXML(DataSet: TDataSet; FileName: string);
…
```

No *Delphi Prism* (e também C#) podemos usar um excelente recurso chamado *extension methods* para resolver o problema da forma mais elegante possível. Podemos **injetar** funcionalidade em uma classe existente, sem usar herança, nem *class helpers* (na verdade isso é muito semelhante a um *class helper*, do *Delphi 8*). A técnica é simples, criamos uma classe estática, decorada com o atributo *Extension* e colocamos um método também decorado com o mesmo atributo indicando qual classe queremos injetar o código. A partir daí, todos os locais que estiverem usando a classe alvo passam a enxergar o código injetado como se ele fosse feito na própria classe! É como se pudéssemos ter os fontes da classe e alterar ao nosso gosto.

Vamos a um exemplo real, que utilizei em um projeto recentemente. O ADO.NET contém a classe *DataSet*, que representa uma estrutura de dados em memória, semelhante ao *TClientDataSet* da VCL. O *DataSet*, assim como o CDS, pode ser representado em formato XML, porém, não usa a mesma estrutura especificada pelo *DataSnap*, os *DataPackets*. Então, criei um método que transforma estruturas *DataSets* em *DataPackets* do MIDAS, o que permite a troca de dados entre aplicações ASP.NET e servidores *DataSnap*, por exemplo, e vice-versa. A solução mais elegante que encontrei foi injetar na própria classe *DataSet* do *.NET Framework* o meu método personalizado. A partir daí, eu poderia usar o código em qualquer lugar, em qualquer projeto, acessando a própria classe *DataSet*. A **Listagem 12** mostra como fazer isso. Note, no exemplo de uso ao final da listagem, que o método pertence à própria classe do *framework*, como se tivéssemos de posse dos seus fontes e pudéssemos alterá-los.

Listagem 12. Estendendo uma classe sem usar herança

```
[Código da classe de extensão]
namespace;
interface

uses
  System,
  System.Data,
  System.Runtime.CompilerServices,
  System.Xml;

type
  [Extension]
  ExtendDataSet = public static class
  public
    [Extension]
    class method ToMidas(s: DataSet): XmlDocument;
  end;

implementation

class method ExtendDataSet.ToMidas(s: DataSet): XmlDocument;
begin
    // aqui vai a lógica da transformação
end;

end.

[Código no Web Form]

method _Default.TestandoDataSetEstendido();
begin
  var ds := new DataSet();
  // aqui você obtém dados para o DataSet
  // Gera um DataPacket
  // Repare que ToMidas é um método injetado
  var MyXmlDataPacket := ds.ToMidas();
  // faz algo com o DataPacket
end;
```

Conclusão

Desde a primeira parte do artigo, evoluímos muito os nossos conhecimentos em orientação a objetos com o *Delphi Prism*. Sem dúvida, a principal barreira é quebrar preconceitos. A rigidez do *Pascal* perdeu forças, no entanto, continuamos programando usando excelentes práticas e OO em alto nível. Lembre-se, novamente, do que disse logo no início deste curso: você precisa abrir a mente para aprender o *Prism*. Não veja as coisas como imutáveis e eternas.

Neste artigo, tocamos em um ponto delicado da OO: classes. Vimos que tipos podem nem mesmo ter um nome, podem ser totalmente declarados e inicializados em uma única linha. Classes parciais parecem coisa de outro mundo, ou POG. Mas não, como vimos, o próprio *.NET Framework* tira excelente proveito do recurso. E finalmente, vimos que injetar código em classes pré-existentes vai melhorar a legibilidade e tornar nosso código muito, mas muito mais OO, sem precisar recorrer à programação estruturada. Bem-vindos a um novo mundo e até a próxima edição. ●

Links

How to create a generic List of anonymous types?
kirillosenkov.blogspot.com/2008/01/how-to-create-generic-list-of-anonymous.html

Exportando DataSets para XML, TXT, Word, Excel, HTML
www.devmedia.com.br/articles/viewcomp.asp?comp=13629

Página do autor na Embarcadero com inúmeros projetos em Delphi e Turbo Pascal
cc.embarcadero.com/Author/222668

CLS
msdn.microsoft.com/en-us/library/12a7a7h3(VS.71).aspx

IL / MSIL
en.wikipedia.org/wiki/Common_Intermediate_Language

Blog do autor
http://guintherpauli.blogspot.com
http://twitter.com/guintherpauli

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

POO – Crie sistemas flexíveis, expansíveis e com baixo custo de manutenção

Programando para interfaces, usando polimorfismo e padrões de projetos

Do que trata o artigo
Fundamentos de OO: abstração, encapsulamento, polimorfismo e herança. Mostra como criar e utilizar interfaces, separar responsabilidades e aplicar padrões de projeto (Design Patterns)

Para que serve
Aumentar o conhecimento sobre POO para desenvolver sistemas mais flexíveis com baixo custo de manutenção, tornando a arquitetura de sistemas mais expansível

Em que situação o tema é útil
As boas práticas apresentadas neste artigo são úteis ao desenvolver aplicações Delphi de qualquer tipo, sejam aplicações comerciais, desktop, web, pois a OO pode ser empregada com sabedoria em qualquer situação, tornando a arquitetura mais robusta

Resumo do DevMan

Este artigo mostra para você o caminho a ser seguido para criar sistemas flexíveis que tenham baixo custo de manutenção. Aqui não será discutido, o que é uma abstração ou uma interface, mais sim como, quando, porque e qual a melhor maneira de usar este e outros recursos.

Rafael Stavarengo
faelsta@gmail.com
É analista e programador da Cheina Informática. Trabalha com Delphi há sete anos, e Java desenvolvendo aplicações cliente/servidor e JME. Formando em Técnologia de Análise e Desenvolvimento de Sistemas pela UNIPAR.
http://www.devmedia.com.br/stavarengo

Programar orientado a objeto não é difícil. É tão fácil que até os programadores mais inexperientes programam utilizando a orientação a objetos, antes mesmo de saber que existe esta forma de programar. Isto pode ser notado, por exemplo, ao criar um novo formulário. Quando você pede um novo formulário (*File + New + Form*), o que acontece de verdade é que o IDE cria automaticamente para você, uma classe que herda as características necessárias para um formulário. É esta facilidade que gera alguns problemas.

Devido à grande facilidade, que permite ao programador codificar sem a necessidade de seguir um padrão específico, os fundamentos da orientação a objetos muitas vezes são ignorados e quando são utilizados, acabam por ser utilizados de maneiras pouco eficientes, levando o programador a pensar que: ou ele não sabe programar orientado a objeto, ou programar orientados a objeto é muito difícil e não vale a pena o tempo gasto em relação aos benefícios que traz. Não importa o que você pensa, o fato é que querendo ou não, sabendo ou não, em *Object Pascal* você sempre está programando OO. O "segredo" é saber usar os fundamentos da OO da melhor maneira possível para o seu caso.

O foco deste artigo não é apresentar os fundamentos da OO, mas sim abordar a melhor maneira de usá-los, aplicando alguns dos principais princípios da programação OO para que o resultado seja um programa flexível, como a nossa realidade exige.

O Jogo

Para entender quais as vantagens e desvantagens da OO, imagine que você criou um jogo que simula o desempenho de diferentes tipos bolas em ambientes diferentes. Todas as bolas do jogo podem rolar ou quicar. Neste jogo, você usou a herança para reduzir ao máximo a duplicação de código. Você criou uma superclasse *TBola* herdada por todos os outros tipos de bolas (**Figura 1).**

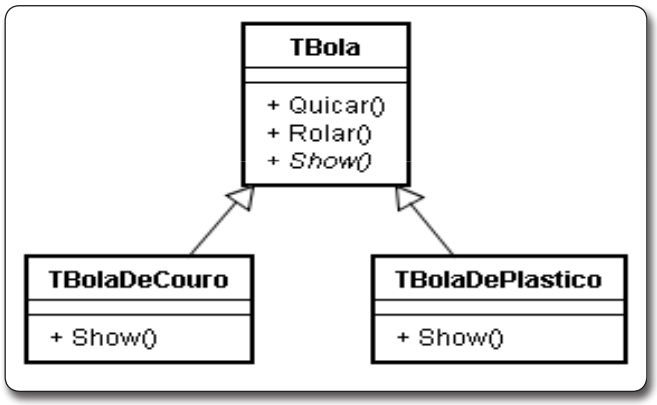


Figura 1. Primeiro design para o simulador (framework)

Na **Figura 1** podemos observar que:

• Além de *TBolaDeCouro* e *TBolaDePlastico* **podem existir outros tipos de bolas que herdam de TBola**;

• Como todas as bolas podem **quicar e rolar da mesma maneira**, a superclasse *TBola* cuida da implementação para os métodos *Quicar()* e *Rolar()*. Fazendo isto, você aproveita uma das vantagens da herança que é o reaproveitamento de código. Implementando os métodos *Quicar()* e *Rolar()*, apenas em uma classe é o suficiente para que todas as suas subclasses tenham o mesmo comportamento;

• Como cada bola tem um desenho e formato diferente – umas com quadradinhos desenhados, outras sem nada desenhado e assim por diante – o método *Show()* é abstrato e cada subclasse deve implementar a maneira como é exibida.

As alterações começam a chegar

O simulador está estável e funciona perfeitamente, porém os concorrentes sempre estão inovando e agora o simulador precisa de novas funções para continuar líder no mercado. Após alguma análise, é decidido que o simulador deve esvaziar as bolas e essa nova função precisa estar pronta em uma semana. Parece que uma semana é tempo de sobra já que esta é uma alteração simples. Basta aproveitarmos mais uma vez da herança criada e adicionar um novo método *Murchar()* na superclasse *TBola* e todas as suas descendentes poderão murchar também (**Figura 2)**.

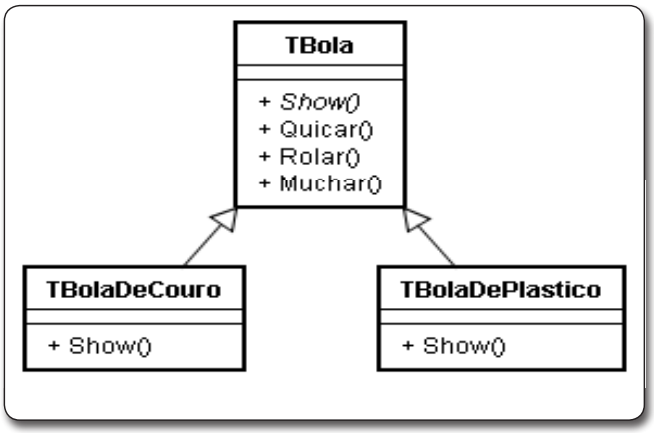


Figura 2. Método Murchar() na classe base

Implementando o método murchar em *TBola*, você evita reescrever o mesmo código em cada classe de bola e consegue terminar a alteração e menos de uma semana, como havia prometido. Se futuramente a forma como as bolas murcham precisar ser alterado, basta fazer uma única alteração.

Usar a herança e implementar o método *Murchar()* na superclasse *TBola* economizou tempo, mas por outro lado criou um comportamento estranho no simulador, pois algumas bolas de boliche e de sinuca começaram a esvaziar durante o jogo. E agora? Como você resolve este problema? Sobrescrever o método *Murchar()* em cada subclasse de *TBola* parece uma boa solução, mas desta maneira você terá espalhado por todo o programa o código que faz uma bola murchar e com certeza o tempo gasto em futuras alterações seria monstruoso. Sem contar a grande porta que se abre para os erros, já que quando você precisar alterar o método *Murchar()*, poderá acabar se esquecendo de mudá-lo em uma classe ou outra.

Você sabe que com o passar do tempo muitos outros tipos de bolas irão surgir e você não pode prever o material e nem o formato que elas terão. É possível que algumas delas tenham um formato diferente de redondo e por isto, rolam de maneira diferente e dependendo do material que é feita, também não deveria murchar. Você precisa de uma solução mais simples, para que somente as bolas que podem esvaziar implementem o método *Murchar()* e ainda mais, que este método de murchar seja escrito somente uma vez. Será que existe um padrão de projeto para isto?

Não misture as bolas

Até agora vimos que herança tem suas vantagens, como evitar a duplicação de código, mas por outro lado, fazer com que todas as subclasses de uma superclasse herdem todos os seus comportamentos não parece muito bom, pois uma simples alteração na superclasse pode fazer com que as subclasses tenham comportamentos estranhos como foi o caso das bolas de sinuca e boliche. Isto torna a manutenção do software um verdadeiro pesadelo.

Para solucionar este problema, primeiro vamos identificar o que muda de uma bola para outra. No nosso simulador o que muda são os comportamentos de esvaziar e rolar, já que nem todas as bolas esvaziam ou rolam e até mesmo entre as que rolam existem diferença, pois uma bola de futebol americano não roda da mesma maneira que uma bola de basquete, por exemplo. Portanto, o que muda em nosso código são os comportamentos de rolar e esvaziar e a classe *TBola* não deve implementá-los.

Sabendo disto, podemos separar as bolas do simulador em dois grupos: bolas que rolam e bolas que murcham. Assim você está aplicando seu primeiro princípio de orientação a objetos.

Princípio de orientação a objetos

Identifique os aspectos que mudam em seu aplicativo e separe-os dos que permanecem iguais.

Pelo que vimos até agora, só temos problemas com os métodos *Murchar()* e *Rolar()*, os demais métodos não variam entre as subclasses de *TBola* e estão funcionando muito bem. Por tanto vamos deixar um pouco a classe *TBola* de lado e nos concentrar nos grupos de classes que criamos.

Para separar os comportamentos que mudam entre uma bola e outra, vamos retirar os métodos *Murchar()* e *Rolar()* de *TBola* e colocá-los em grupos específicos de classes que representam estes comportamentos **(Figura 3)**.

Nota

Não se preocupe como são feitos este grupo de classes. Você verá isto em outro momento.

Definido os grupos de comportamentos, queremos agora que as subclasses de *TBola* possam rolar e esvaziar sem perder a flexibilidade do projeto. Por exemplo, queremos fazer uma bola de basquete rolar e após isto acontecer, gostaríamos de trocar a maneira como ela rola ou até mesmo fazer com que ela não possa rolar mais, tudo isso em tempo de execução. Em outras palavras, queremos alterar a maneira como uma bola rola ou murcha sem precisar re-compilar o simulador.

Princípio de orientação a objetos

Programe para uma interface, não para uma implementação.

Vamos aplicar o princípio da orientação a objetos que nos diz para SEMPRE programar para *interface* (contrato), ou seja, devemos sempre identificar os grupos de classes com

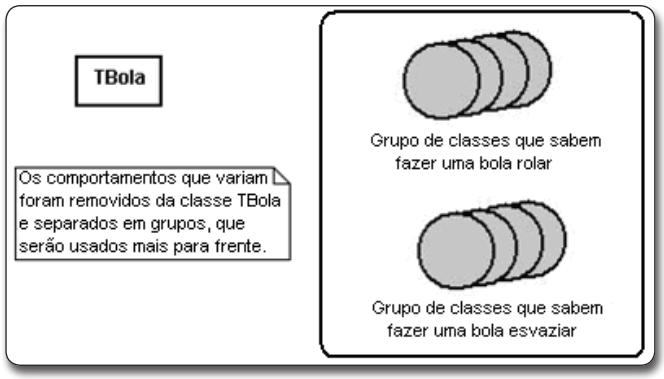


Figura 3. Separando o que muda

comportamentos em comum e criar *interfaces* que identifiquem o que elas têm em comum. No nosso simulador, vamos criar duas *interfaces*: *IComportamentoDeRolar* e *IComportamentoDeMurchar* e assim qualquer classe que queira rolar ou murchar basta implementar a *interface* que identifica o comportamento desejado. Desta maneira, ao invés das subclasses de *TBola* implementar os métodos *Rolar()* e *Murchar()*, existirá um grupo de classes cuja única razão de existir é o de representar um destes comportamentos (**Figura 4**).

Na **Figura 4** podemos ver as classes de comportamentos, cujo o único objetivo de existir é implementar o método que a *interface* ao qual ela implementa declara. Algumas considerações:

- Cada classe que implementa a *interface IComportamentoDeRolar* deve escrever o código necessário para fazer uma bola rolar;
- A classe *TRolarComoBolaDeFutebolAmericano* faz com que uma bola role como se fosse uma bola de futebol americano;
- A classe *TNaoPodeRolar* deixa o corpo do método *Rolar()* vazio, visto que as bolas com este comportamento não devem se movimentar.

Veja nas **Listagens 1**, **2** e **3** como ficam os códigos destas classes.

Listagem 1. Declarando a interface para classes que sabem rolar uma bola

```
unit ComportamentoDeRolar;

interface

type
  IComportamentoDeRolar = interface
    ['{83D1C793-73D3-4F22-83DF-FCF51905B815}']
    procedure Rolar;
  end;

implementation

end.
```

Na **Listagem 1** temos a declaração da *interface* (contrato) que as classes devem implementar (assinar), para poder fazer parte do grupo de classes que sabem rolar uma bola.

Na **listagem 2** temos a classe que sabe fazer uma bola rolar como se fosse uma bola de futebol americano.

Nota

Estes arquivos de código-fonte serão criados adiante neste exemplo.

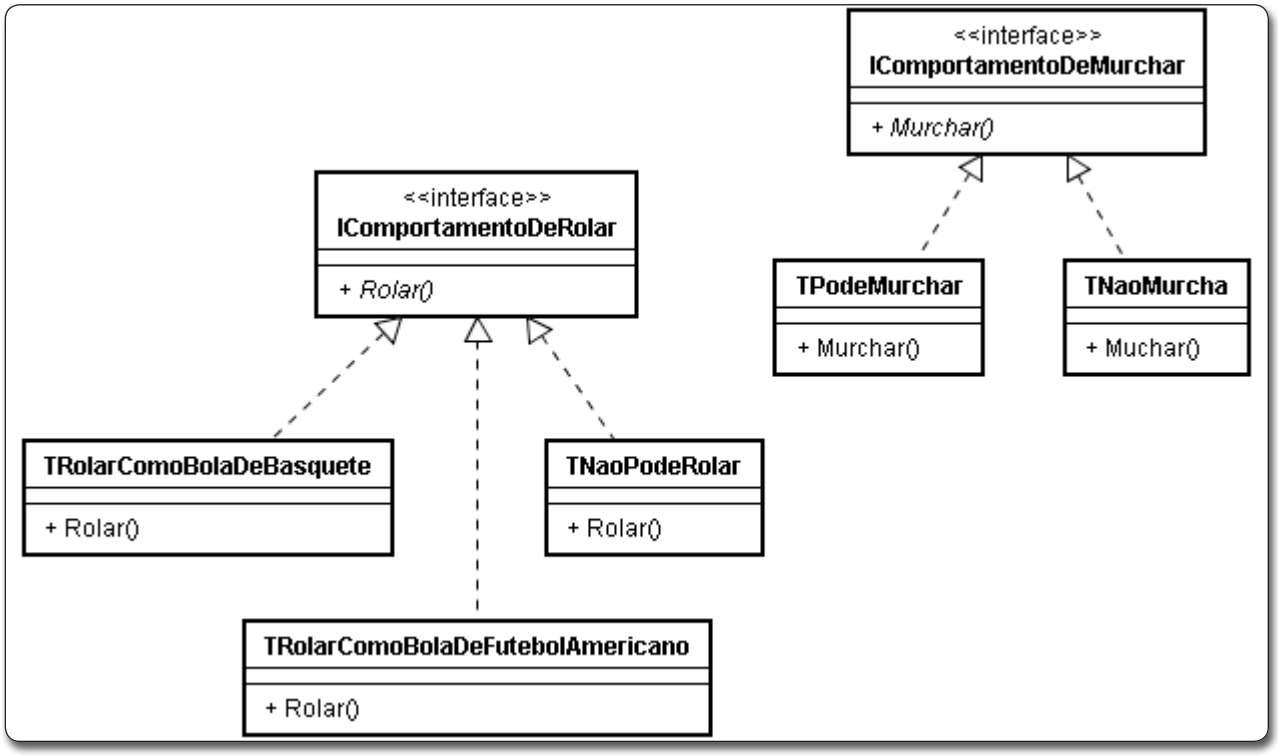


Figura 4. Criando as classes de comportamentos

Na **Listagem 3** vemos a criação da classe *TNaoPodeRolar*. Usaremos uma instância desta classe para fazer com que um objeto *TBola* pare de rolar. A classe *TNaoPodeRolar* não implementa código algum no método *Rolar()* pois ela não deve mover as bolas em nosso simulador.

Do jeito que o simulador foi feito no início, estávamos contando com uma implementação feita na superclasse e por isso ficávamos presos a uma única definição de comportamento e se quiséssemos fazer alterações neste comportamento deveríamos escrever mais e mais código a cada novo tipo de bola que fosse inventada. Agora estamos usando um contrato (as *interfaces*) que definem regras para qualquer classe que deseje assiná-lo. Isto pode parecer um pouco radical a princípio.

Nota do DevMan

Se você não tem muita experiência com as interfaces do Object Pascal, deve ter achado estranho o código ['{83D1C793-73D3-4F22-83DF-FCF51905B815}'], que você pode ver na Listagem 1. Este código é chamado de GUID - Globally Unique Identifier e serve para tornar sua interface "única no mundo". Tudo o que você precisa saber sobre ele é que não é obrigatória sua declaração para criar uma interface, porém, você ganha uma série de facilidades ao declará-lo. Fora as facilidades com type cast, você também ganha a capacidade de testar se uma classe implementa ou não uma interface, como no código a seguir: if Supports(Obj, IComportamentoDeRolar) then. Este código testa se objeto Obj implementa a interface IComportamentoDeRolar. Se você não declarar o GUID na interface IComportamentoDeRolar, este teste não funcionara. O IDE pode gerar um GUID aleatório para você, basta pressionar Ctrl + Shift + G.

Listagem 2. Criando a classe TRolarComoBolaDeFutebolAmericano

```
unit RolarComoBolaDeFutebolAmericano;

interface

uses ComportamentoDeRolar;

type
  TRolarComoBolaDeFutebolAmericano = class(TInterfacedObject,
   IComportamentoDeRolar)
  public
    procedure Rolar;
  end;

implementation

uses Dialogs;

procedure TRolarComoBolaDeFutebolAmericano.Rolar;
begin
  ShowMessage('Estou rolando como uma bola de futebol americano');
  {aqui vai o código necessário para fazer uma bola rolar
   como se fosse uma bola de futebol americano}
end;

end.
```

Listagem 3. Criando a classe TNaoPodeRolar

```
unit NaoPodeRolar;

interface

uses ComportamentoDeRolar;

type
  TNaoPodeRolar = class(TInterfacedObject, IComportamentoDeRolar)
  public
    procedure Rolar;
  end;

implementation

uses Dialogs;

procedure TNaoPodeRolar.Rolar;
begin
  //Sou uma bola que não pode rolar
end;

end.
```

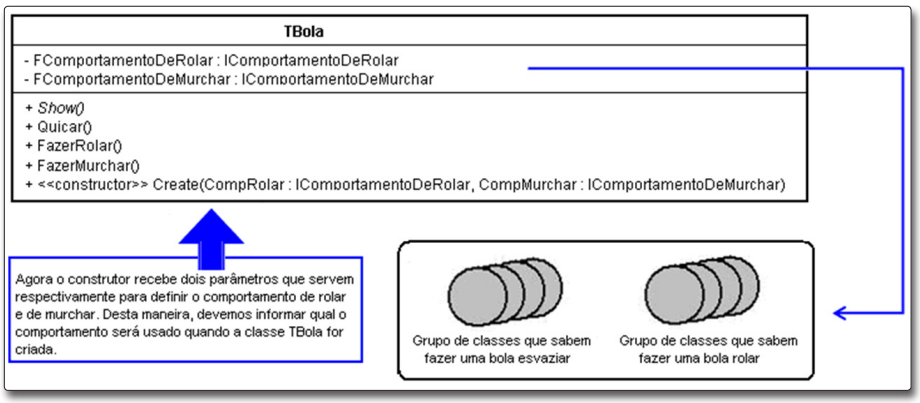


Figura 5. Novo design de TBola

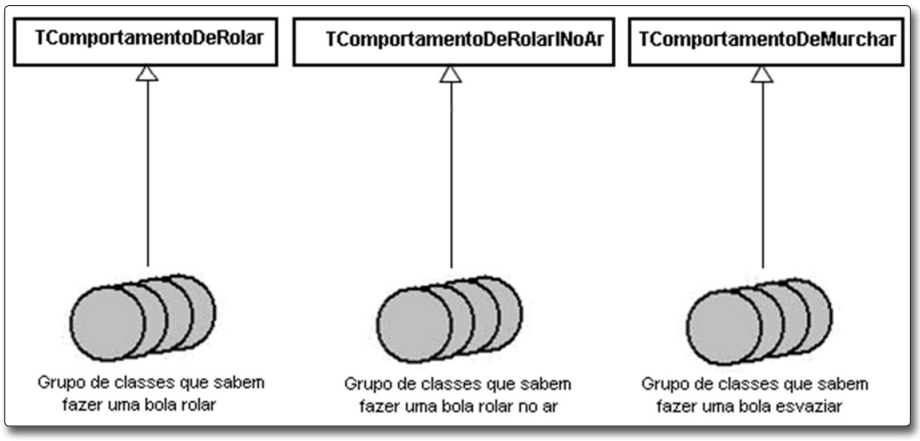


Figura 6. Programando para uma implementação ao invés de interface

Programar para uma *interface* significa na verdade, programar para um contrato. A vantagem está na arquitetura, pois através das *interfaces*, podemos referenciar um objeto sem saber realmente qual o seu tipo, pois o que importa é se ele implementa aquela *interface* ou não.

Mudando o comportamento em tempo de execução

Agora que você entendeu as vantagem e desvantagens da herança e mudou a maneira de pensar quando se deseja obter mais flexibilidade no sistema, vamos fazer o simulador alterar o comportamento das bolas em tempo de execução. Veja na **Figura 5** o que muda na classe *TBola* após aplicados os princípios aprendidos no tópico anterior.

Princípio de orientação a objetos

Dê prioridade à composição.

A **Figura 5** mostra uma mudança sutil, porém, reveladora em *TBola*. Agora, cada bola **TEM UM** *IComportamentoDeRolar* e um *IComportamentoDeMurchar*. Relacionamentos deste tipo entre os objetos recebem o nome de composição. No decorrer do artigo você entenderá as vantagens de TER UM (composição), sobre SER UM (herança).

Foram feita três mudanças na classe *TBola*. Primeiro foram adicionadas duas novas variáveis que definem o comportamento que uma bola terá em tempo de execução. Repare que estas variáveis são do tipo das *interfaces* criadas para definir os grupos de comportamentos. Desta maneira as variáveis não ficam presas a uma implementação e assim elas poderão referenciar qualquer tipo de objeto existente, desde que estes objetos implementem a *interface* pedida. Se você definisse a variável *FComportamentoDeRolar* do tipo de uma classe que faz rolar, como por exemplo *TRolarComoBolaDeBasquete*, esta bola só poderia rolar como uma bola de basquete, pois a variável só aceitaria instâncias de objetos da classe *TRolarComoBolaDeBasquete*.

Então você deve estar se perguntando: "Por que eu não posso criar uma superclasse para os grupos de classes de comportamentos ao invés de criar uma *interface*?". A resposta é: você pode, mas isto seria ruim. A grande vantagem da *interface* em relação à herança é que uma classe pode implementar quantas interfaces você quiser, enquanto a herança só pode ser feita de uma única classe. Você estaria programando para implementações novamente e se no futuro você precisar fazer as bolas rolarem em quanto estiverem quicando, teria que criar uma nova superclasse para este novo comportamento de rolar. Veja como ficaria na **Figura 6**.

Na **Figura 6** a herança mostra mais uma vez que sozinha não consegue resolver todos os problemas, deixando o programa inflexível. Além dos problemas já vistos nos tópicos anteriores, temos o grande número de classes novas criadas em duplicidade

apenas para o novo comportamento de rolar no ar, isto porque uma classe não pode herdar mais de uma classe ao mesmo tempo (o *Delphi* não suporta herança múltipla). Programando para *interfaces*, você não tem este problema, pois uma mesma classe pode implementar quantas interfaces for necessário, em outras palavras, uma mesma classe pode assinar quantos contratos você quiser (isso **simula** herança múltipla).

A **Figura 7** mostra que não precisamos criar mais classes para o caso de implementar um novo comportamento de rolar. Neste caso, seria necessário apenas criar mais uma *interface* para este comportamento e implementá-la nas classes já existentes.

Voltando para a classe *TBola*, vamos definir o comportamento que queremos para a bola no momento em que criarmos ela, por isto foram adicionados dois parâmetros ao construtor da classe. Por último, foi alterado o nome dos métodos *Rolar()* e *Murchar()* para *FazerRolar()* e *FazerMurchar()*, respectivamente. Os nomes foram alterados porque agora os métodos que fazem uma bola rolar e murchar pertencem às classes de comportamentos – aquelas que implementam as classes *IComportamentoDeMurchar* ou *IComportamentoDeRolar*. Como estes métodos não pertencem mais a classe *TBola*, então faz mais sentido chamar o método de *FazerRolar()* do que apenas *Rolar()*. Nada impede que você use os nomes antigos. Nenhum problema será criado por conta disto, apenas fica mais fácil de ler e entender o código com os novos nomes.

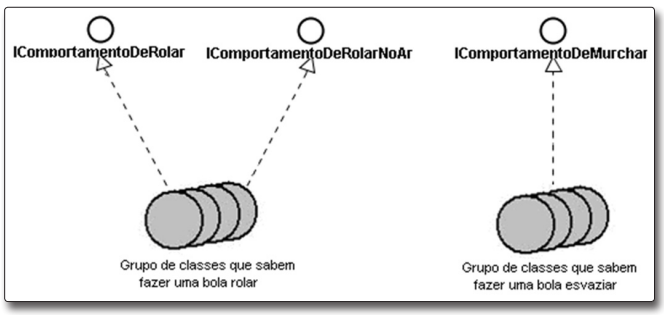


Figura 7. Programando para interface

Da teoria à prática

Feitas todas as alterações necessárias no design inicial do simulador, chegou a hora de colocar tudo isto para funcionar. Neste tópico, você tem a oportunidade de praticar os conceitos abordados neste artigo.

Não existe muita diferença entre a manipulação das classes que fazem uma bola rolar e manipulação das classes que fazem murchar, por isto, neste exemplo vamos trabalhar apenas com as classes que definem o comportamento de rolar, porém, no exemplo disponível para download, todos os comportamentos são manipulados. Recomendo, de verdade, que você tente fazer sozinho toda a parte referente ao comportamento de murchar e só depois de ter conseguido, use o exemplo para conferir se você fez algo diferente. Mas primeiro, será feita a manipulação do comportamento de rolar.

Crie um novo projeto (*File + New + Application*), deixe o formulário principal como na **Figura 8**. Após isto, configure os componentes do formulário principal de acordo com a **Tabela 1**.

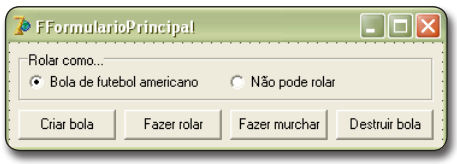


Figura 8. Formulário principal do exemplo

Objeto	Propriedade	Valor
TForm	Name	FFormularioPrincipal
TRadioGroup	Name	RGRolarComo
	Caption	Rolar como...
	Columns	2
	ItemIndex	0
	Items	Bola de futebol americano Não pode rolar
TButton	Name	BCriar
	Caption	Criar bola
TButton	Name	BFazerRolar
	Caption	Fazer rolar
	Enabled	False
TButton	Name	BFazerMurchar
	Caption	Fazer murchar
	Enabled	False
TButton	Name	BDestruirBola
	Caption	Destruir bola
	Enabled	False

Tabela 1. Componentes do formulário principal

Listagem 4. Evento OnClick do botão BCriar

```
1 procedure TFFormularioPrincipal.BCriarClick(Sender: TObject);
2 Var
3   CompDeRolar: IComportamentoDeRolar;
4 Begin
5   case RGRolarComo.ItemIndex of
6     0: CompDeRolar := TRolarComoBolaDeFutebolAmericano.Create;
7     1: CompDeRolar := TnaoPodeRolar.Create;
8   End;
9
10   FBola := TBola.Create(CompDeRolar, nil);
11
12   BCriar.Enabled := False;
13   BFazerRolar.Enabled := True;
14   BFazerMurchar.Enabled := True;
15   BDestruirBola.Enabled := True;
16   RGRolarComo.Enabled := False;
17 end;
```

Na **Listagem 4** você pode ver como a classe *TBola* é criada dentro do simulador.

- Na linha 3, declaramos a variável que vai receber uma referência a uma classe que implemente a *interface IComportamentoDeRolar*. Não esqueça de declarar a unit *ComportamentoDeRolar* na sessão *uses* do formulário principal;
- Entre as linhas 5 e 8, é feito um teste para saber qual comportamento de rolar deve ser usado. Observe que, para os dois comportamentos é usado a mesma variável, embora as classes que estão sendo criadas sejam diferentes. Isto porque o sistema foi programado para *interface* e a variável da linha 3 é do tipo de uma *interface* e não um tipo de uma classe (implementação). Não esqueça de declarar as *units RolarComoBolaDeFutebolAmericano* e *NaoPodeRolar* na sessão *uses* do formulário principal;

• Na linha 10, a *TBola* é criada definitivamente e no seu construtor passamos a variável que possui uma referência para o comportamento que deve ser usado. É passado **nil** para o parâmetro que define o comportamento de murchar, porque não vamos manipular este comportamento neste exemplo;
• A variável *FBola* usada na linha 10 foi declarada (*FBola: TBola*), na sessão *private* do formulário. Não esqueça de declarar a *unit Bola* na sessão *uses* do formulário principal;
• Com a bola criada, o botão de criar bolas é desabilitado e os demais botões que manipulam uma bola são habilitados, como pode ser visto entre as linhas 12 e 16.

Listagem 5. Evento OnClick dos botões BFazerRolar, BFazerMurchar e BDestruirBola

```
1 procedure TFFormularioPrincipal.BFazerRolarClick(Sender: TObject);
2 Begin
3   FBola.FazerRolar;
4 end;
5
6 procedure TFFormularioPrincipal.BFazerMurcharClick(Sender: TObject);
7 Begin
8   FBola.FazerMurchar;
9 end;
10
11 procedure TFFormularioPrincipal.BDestruirBolaClick(Sender:
     TObject);
12   Begin
13    FreeAndNil(FBola);
14
15    BCriar.Enabled := True;
16    BFazerRolar.Enabled := False;
17    BFazerMurchar.Enabled := False;
18    BDestruirBola.Enabled := False;
19    RGRolarComo.Enabled := True;
20   end;
```

Não há segredos na **Listagem 5.**
• Da linha 1 a 4 é definido o evento *OnClick* do botão *BFazerRolar*. Quando o usuário clicar neste botão a bola deve rolar de acordo com o comportamento que ele escolheu antes de criar a bola;
• Da linha 6 a 9 é definido o evento *OnClick* do botão *BFazerMurchar*. Embora não tenha sido definido um comportamento de murchar quando a classe *TBola* foi criada, nada impede o formulário principal de chamar o método *FazerMurchar()*, isto porque quem deve se preocupar se o comportamento foi definido ou não é a classe *TBola* e não o formulário principal;
• Entre as linhas 11 e 20 você pode ver o evento *OnClick* do botão *BDestruirBola*. Quando o usuário clicar neste botão a instância da bola que foi criada é destruída e os objetos da tela são atualizados para que o usuário possa criar uma outra bola.

Na **Listagem 6** você pode ver como a classe *TBola* é por dentro. Para criar a classe *TBola*, peça uma nova unit (*File + New + Unit*), salve-a com o nome de *Bola.pas* e digite o código apresentado. Explicando:
• Nas linhas 10 e 11 são declaradas as variáveis que receberão o respectivo comportamento escolhido quando a classe é criada;
• Nas linhas 26 e 27 é atribuído o valor para as variáveis que definem o comportamento que a bola deve ter;
• Entre as linhas 38 e 42 é feita a implementação do método *FazerRolar()*. Este método apenas chama o método *Rolar()* da variável *FComportamentoDeRolar* definida na linha 10. Repare na linha 40 que existe um teste para garantir que o método *Rolar()* só será invocado quando tiver um comportamento definido.

Listagem 6. A classe TBola

```
1 unit Bola;
2
3 Interface
4
5 uses ComportamentoDeRolar, ComportamentoDeMurchar;
6
7 Type
8   TBola = class
9   Private
10    FComportamentoDeRolar: IComportamentoDeRolar;
11    FComportamentoDeMurchar: IComportamentoDeMurchar;
12  Public
13    constructor Create(CompRolar: IComportamentoDeRolar;
       CompMurchar: IComportamentoDeMurchar);
14    procedure Show;
15    procedure Quicar;
16    procedure FazerRolar;
17    procedure FazerMurchar;
18  End;
19
20 Implementation
21
22 { TBola }
23
24 constructor TBola.Create(CompRolar: IComportamentoDeRolar;
     CompMurchar: IComportamentoDeMurchar);
25 Begin
26   FComportamentoDeRolar := CompRolar;
27   FComportamentoDeMurchar := CompMurchar;
28
29   Show;
30 End;
31
32 procedure TBola.FazerMurchar;
33 Begin
34   if Assigned(FComportamentoDeMurchar) then
35     FComportamentoDeMurchar.Murchar;
36 end;
37
38 procedure TBola.FazerRolar;
39 Begin
40   If Assigned(FComportamentoDeRolar) then
41     FComportamentoDeRolar.Rolar;
42 end;
43
44 procedure TBola.Quicar;
45 Begin
46   //código para fazer uma bola quicar
47 end;
48
49 procedure TBola.Show;
50 Begin
51   //desenha uma bola na tela
52  end;
53
54 end.
```

Isto vai evitar os erros de acesso à memória (*Access Violation*), quando nenhum comportamento de rolar for definido;
• Entre as linhas 32 e 36 está a chamada do método *Murchar()* da variável que define o comportamento de esvaziar uma bola. Lembre-se que as classes que definem este comportamento devem ser criadas por você. Após criá-las, você pode baixar o arquivo de exemplo para conferir se você fez algo diferente.

Para criar a interface *IComportamentoDeRolar*, peça uma nova unit (*File + New + Unit*), salve-a com o nome de *ComportamentoDeRolar.pas* e digite o código da **Listagem 1**. Para criar a classe *TRolarComoBolaDeFutebolAmericano*, peça uma nova unit (*File + New + Unit*), salve-a com o nome de *RolarComoBolaDeFutebolAmericano.pas* e digite o código da **Listagem 2**. Para criar a classe *TNaoPodeRolar*, peça uma nova unit (*File + New + Unit*), salve-a com o nome de *NaoPodeRolar.pas* e digite o código da **Listagem 3**. Após seguir estes passos, ainda estará faltando criar a *interface IComportamentoDeMurchar*. Crie uma nova unit (*File + New + Unit*), salve-a com o nome *ComportamentoDeMurchar.pas* e digite o código **Listagem 7**.

Listagem 7. Declarando a interface IComportamentoDeMurchar

```
unit ComportamentoDeMurchar;

interface

type
  IComportamentoDeMurchar = interface
    ['{9A434BD8-7F13-43EF-BB28-9760137DD94B}']
    procedure Murchar;
  end;

implementation

end.
```

Veja que o código da **Listagem 1** e da **Listagem 7** são muito parecidos, isto porque ambas as listagens estão definindo as *interfaces* para os grupos de classes de comportamentos. Após criar todos os arquivos, você pode compilar e executar a aplicação para ver o resultado.

Lembrete!

Depois que você ver o resultado, não se esqueça de criar as classes que definem o comportamento de esvaziar e usá-las assim como já esta feito para o comportamento de rolar. Uma dica: Estas classes devem implementar a interface IComportamentoDeMurchar.

Nota do DevMan

A função Assigned(const P) usada na Listagem 6 é uma função nativa do Delphi. Está função recebe uma variável que aponta para um objeto ou método e testa se o local para onde a variável aponta é válido. Se o ponteiro passado no parâmetro P estiver apontando para um local válido é retornado True, caso o ponteiro esteja apontando para um endereço inválido é retornado False. Quando Obj estiver apontando para um objeto, usar if Assigned(Obj) then é o mesmo efeito que usar if Obj <> nil then. Quando Obj estiver apontando para um método, usar if Assigned(Obj) then será o mesmo efeito que usar if @Obj <> nil then.

Ainda mais flexível

Já criou as classes que definem o comportamento de murchar? Então vamos continuar. Aplicando alguns princípios de programação orientada a objetos, foi possível tornar o simulador muito mais flexível. No primeiro design do simulador não existia espaço para alterações e mudar o comportamento de uma bola era uma tarefa complicada, pois para cada novo comportamento, era necessário criar um novo método na superclasse *TBola*, fazendo com que esta alteração se propagasse para todas as subclasses causando erros e comportamentos estranhos, como por exemplo, fazer uma bola de boliche esvaziar. Com o novo design do simulador podemos definir o comportamento de qualquer objeto da classe *TBola* em tempo de execução quando criamos uma nova bola. Veja na **Figura 9** como ficou o novo design após aplicar os princípios aprendidos até agora.

Na **Figura 9** é possível ver que a classe *TBola* não sabe mais como fazer para esvaziar ou rolar. Ao invés disto, ela invoca os métodos das variáveis *FComportamentoDeRolar* e *FComportamentoDeMurchar*, assim uma bola pode assumir qualquer comportamento em tempo de execução. Para isto, a classe *TBola* depende das *interfaces* que definem os grupos de comportamentos *IComportamentoDeRolar* e *IComportamentoDeMurchar*, assim se no futuro for preciso adicionar ou remover uma nova maneira de murchar ou rolar, basta criar a classe que saberá como este novo comportamento deve ser executado e implementar a *interface* correta. É importante frisar que a classe *TBola* não depende das classes de comportamento, mas sim das *interfaces*. Isto significa que se você excluir ou adicionar algumas das classes de comportamento, a classe *TBola* continuará compilando como se nada tivesse acontecido, pois ela depende apenas das *interfaces*.

Graças às interfaces (e porque não dizer polimorfismo), o simulador alcançou um bom nível de abstração, pois tudo aquilo que muda (comportamento de rolar e murchar), foi separado do que não muda na família de *TBola*. Embora seja possível definir o comportamento de uma bola em tempo de execução, a definição do comportamento fica restrita ao momento de criação da bola. Após criar uma bola, se torna impossível alterar o comportamento da mesma sem destruí-la.

Para conseguir alterar o comportamento de uma bola mesmo depois de criá-la, se faz necessário criar mais dois métodos: um para alterar o comportamento de murchar, e outro para alterar o comportamento de rolar.

Listagem 8. Métodos para alterar o comportamento de TBola após sua criação

```
procedure TBola.SetComportamentoDeMurchar(
  NovoComportamento: IComportamentoDeMurchar);
begin
  FComportamentoDeMurchar := NovoComportamento;
end;

procedure TBola.SetComportamentoDeRolar(
  NovoComportamento: IComportamentoDeRolar);
begin
  FComportamentoDeRolar := NovoComportamento;
end;
```

A **Listagem 8** mostra os métodos *SetComportamentoDeMurchar(IComportamentoDeMurchar)* e *SetComportamentoDeRolar(IComportamentoDeRolar)* que servem respectivamente para alterar o comportamento de uma bola ao executar a ação de esvaziar e rolar. Não á truques nestes métodos. Eles apenas alteram a referência das variáveis de comportamentos que são iniciadas quando *TBola* é criada. Através destes métodos uma bola pode alterar seu comportamento sem precisar ser destruída. Esta facilidade é possível graças ao uso da composição (TEM UM).

Fim do jogo

No primeiro design do simulador (**Figura 1**), a classe *TBola* "era tudo ao mesmo tempo" e não havia espaço para alterações. Ela (a classe *TBola*) era responsável por tudo que todas as bolas deveriam fazer. É claro que isto gerou muitos problemas, pois uma bola pode assumir vários formatos, tamanhos e ainda por cima ser feita de materiais diferentes.

Para melhorar o código do sistema e deixá-lo aberto para alterações foi necessário seguir três princípios de programação orientada a objetos. São eles:

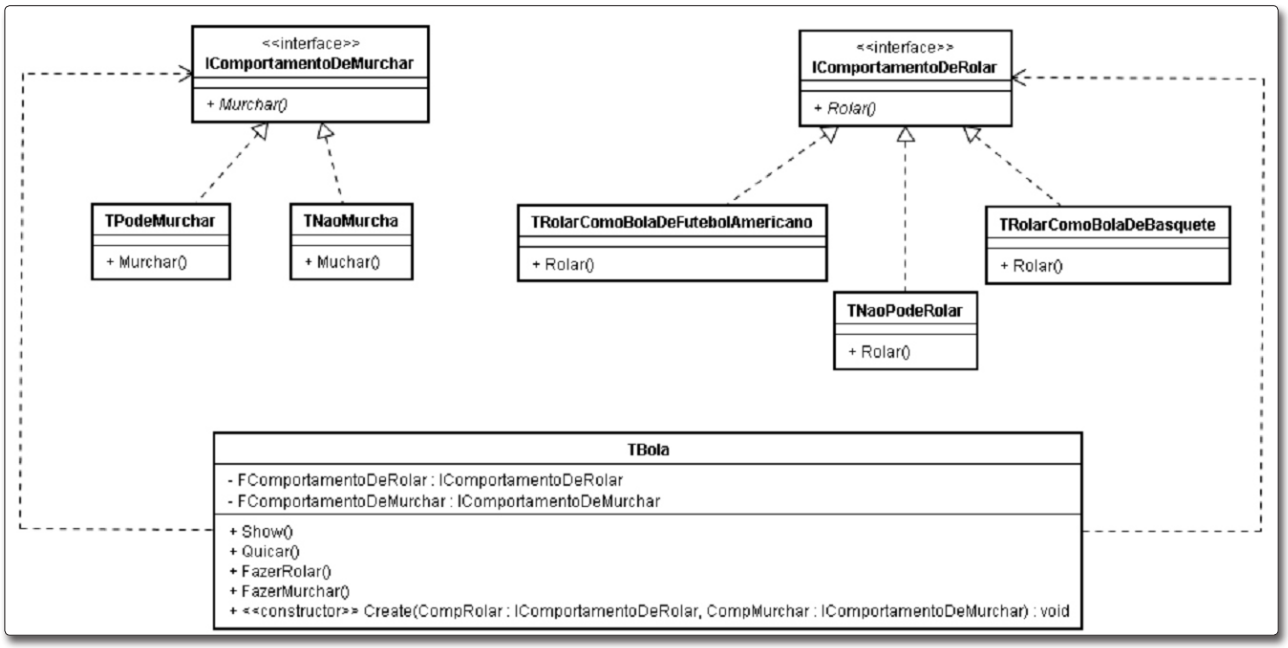


Figura 9. Novo design para o simulador

1. Identifique os aspectos que mudam em seu aplicativo e separe-os do que permanecem iguais.
Quando este princípio foi usado, a classe *TBola* passou a ter menos responsabilidades, pois tudo aquilo que podia ser diferente entre uma bola e outra, como a forma de rolar, por exemplo, foi retirado da superclasse e colocado em outras classes criadas com um único objetivo: fazer uma bola assumir um determinado comportamento. Com esta solução, apareceu outro problema: como saber que tipo de comportamento uma classe representa?

2. Programe para uma interface, não para uma implementação.
Este princípio resolveu dois problemas de uma única vez. Programando para *interfaces* foi possível definir uma maneira de saber qual comportamento de *TBola* uma classe representava, alem de permitir que uma mesma bola pudesse assumir vários tipos de comportamentos diferentes em tempo de execução. Este resultado foi obtido ao usar as *interfaces* para criar regras (ou contratos), que definiam o comportamento de uma bola. Assim qualquer classe que assinasse este contrato seria considerada uma classe de comportamento de *TBola*;

3. Dê prioridade à composição.
Usando composição (TEM UM), uma bola pode mudar o seu comportamento mesmo depois de já estar criada. Desta maneira não é necessário destruir uma bola para alterar o seu comportamento.

Veja que sozinhos estes princípios não trariam um resultado desejável.

Conclusão

Programar orientado a objetos todos os programadores *Object Pascal* já fazem: o que muda de um para o outro é o conhecimento sobre orientação a objetos que cada um tem. Para simplificar o aprendizado, este artigo usou um exemplo de algo incomum para a maioria de nós, mas que pode facilmente ser aplicado em situações mais comuns. Você poderia usar os princípios e fundamentos que aprendeu aqui para abstrair as funções do seu sistema ao emitir um documento fiscal, por exemplo. Todos sabem que existem diversos tipos de documentos fiscais: notas, cupons, notas eletrônicas e assim por diante. Fazendo uma analogia ao exemplo do simulador, a maneira como cada documento fiscal é emitida e armazenada no sistema é a parte que muda. Sabendo disto, é possível criar uma *interface* que identifique este comportamento e fazer com que a classe *TEmiteDocumentoFiscal* (por exemplo), assumisse o comportamento desejado de acordo com o tipo de nota configurado no sistema. ●

Nota do DevMan

Notou que este artigo não faz citações sobre padrões de projeto no texto? Aqui foi discutido apenas alguns dos fundamentos de programação orientado a objetos. Porém mesmo assim, sem perceber, você aprendeu o padrão strategy. Este padrão define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

Em outras palavras: este padrão separa os algoritmos que fazem a mesma coisa de maneira diferente (como rolar uma bola), e encapsula em classes que serão usadas por outra classe denominada Classe Cliente (a classe TBola).

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Introdução à POO – Parte 3

Aprenda, de uma forma simples, a programar da forma correta

Guinther Pauli

guinther.pauli@gmail.com

Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software, dos quais 14 dedicados ao Delphi e 9 ao .NET/C#. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Delphi para Linux, Certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET, VB.NET, MSF, SQL Server). Editor geral da Revista ClubeDelphi, Editor Geral da Revista .NET Magazine, professor de Pós-Graduação. Publicou mais de 150 artigos, 400 vídeo-aulas, palestrante em 5 edições da Borland Conference. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adotação de melhores práticas. Blog: guintherpauli.blogspot.com

Resumo do DevMan

Neste artigo aprenderemos mais sobre métodos virtuais e polimorfismo, dando um passo além, estudando métodos abstratos. Também examinaremos o que são construtores e destrutores. Entendermos o que são e para que servem os especificadores de visibilidade (modificadores) public, protected e private. Criaremos propriedades para nossas classes e reforçaremos princípios básicos de encapsulamento. E finalmente, entenderemos o que são métodos get / set.

Do que trata o artigo

Este artigo aborda a orientação a objetos com o Delphi, usando uma metodologia simples, didática, de fácil aprendizado. Veremos na teoria, e também na prática, todos os conceitos, fundamentos e recursos oferecidos pelo Delphi para promover a POO

Para que serve

A POO pode e deve ser aplicada de forma inteligente, ela serve para construir sistemas mais robustos, mais confiáveis, de fácil manutenção, que permitam maior reaproveitamento de código

Em que situação o tema é útil

A POO é útil em qualquer sistema, seja ele Web, Desktop, Mobile, não importa o tipo de aplicação. Os conceitos aqui apresentados, em um exemplo simples, podem ser utilizados em aplicações reais, como é apresentado em um estudo de caso no final desta série

Nas partes anteriores deste mini-curso, aprendemos importantes técnicas da orientação a objetos com o *Delphi*. Vimos como criar classes, aplicar a herança (*TCarro* e *TAviao* agora herdam de *TMeioTransporte*), vimos como criar métodos virtuais e aplicar o polimorfismo, como sobrescrever métodos virtuais e chamar funcionalidades da classe base com *inherited*. Nesta terceira parte do nosso curso, vamos avançar

mais na orientação a objetos, chegando em um nível muito próximo de aplicações reais, inclusive fazendo novamente comparações com o *design* de classes da *VCL*.

Métodos abstratos

Se lembrar da última parte do curso, implementamos no método *Mover* a funcionalidade básica para todos os meios de transporte, que é o ato de *ligar*. Mas imagine o seguinte: e se cada meio de transporte for ligado de uma forma? O que é comum nesse caso é a chamada do método, ou seja, todos devem ser ligados antes de entrar em movimento. O que muda é a maneira como são ligados (já viu que vamos falar de polimorfismo novamente). Então altere o código do método *Mover* da classe *TMeioTransporte* como mostrado na **Listagem 1.**

Listagem 1. Comportamento básico para todos os meios de transporte

```
procedure TMeioTransporte.Mover();
begin
  Ligar();
end;
```

Isso diz que um meio de transporte deve ser ligado sempre que entrar em movimento (tudo bem que na vida real não seja exatamente assim, mas para fins didáticos, está bom). Agora declare o método *Ligar* na *interface* da classe *TMeioTransporte* (**Listagem 2**).

Listagem 2. Método abstrato

```
unit uMeioTransporte;

interface

type
  TMeioTransporte = class
    Descricao : string;
    Capacidade : integer;
    procedure Mover; virtual;
    procedure Ligar; virtual; abstract;
  end;

implementation

{ TMeioTransporte }

uses
  Dialogs;

procedure TMeioTransporte.Mover;
begin
  ShowMessage('Ligando ' + Descricao);
end;

end.
```

Abstract é sempre usado junto da diretiva **virtual** e indica que o método não possui implementação na classe em que é declarado (algo muito semelhante a uma *interface*, técnicas que discutiremos adiante neste curso). *Ligar* não é implementado nessa classe, ou seja, ele é totalmente abstrato, só possui a definição para poder ser chamado (no método *Mover*). Ou seja, sabemos que todos os meios de transporte são ligados antes de entrar em movimento, então podemos fazer essa chamada na classe base para não precisar fazer isso em todas as classes descendentes. Todos os *TMeioTransporte* são ligados antes de mover, mas são ligados de forma diferente, novamente, polimorfismo. Um carro é ligado com chave, um avião..., bom deixa pra lá.

Usando recursos de polimorfismo vamos sobrescrever o *Ligar* na classe descendente. Abra a classe *TCarro* e sobrescreva o método *Ligar* e o implemente como na **Listagem 3.** Fazemos o mesmo para a classe *TAviao* (**Listagem 4**).

Listagem 3. Sobrescrevendo um método virtual e abstrato: TCarro

```
unit uCarro;

interface

uses
  // coloque essa unit p/ acessar a classe TMeioTransporte
  uMeioTransporte;

type
// observe que TCarro agora herda de TMeioTransporte
  TCarro = class(TMeioTransporte)
    // observe que retiramos os campos Capacidade e Descricao daqui
    Quilometragem : integer;
    procedure Mover(); override;
    procedure Ligar(); override;
  end;

implementation

uses Dialogs;
{ TCarro }

procedure TCarro.Ligar();
begin
  // repare que não vai inherited aqui
  // pois não existe nada na classe base
  ShowMessage('Ligando o carro ' + Descricao);
end;

procedure TCarro.Mover();
begin
  inherited; // isso vai chamar o Ligar
  ShowMessage(Descricao + ' entrou em movimento.');
end;

end.
```

Listagem 4. Sobrescrevendo um método virtual e abstrato: TAviao

```
unit uAviao;

interface

uses
  // coloque essa unit p/ acessar a classe TMeioTransporte
  uMeioTransporte;

type
// observe que TAviao agora herda de TMeioTransporte
  TAviao = class(TMeioTransporte)
    // observe que retiramos os campos Capacidade e Descricao daqui
    HorasVoo : integer;
    procedure Mover(); override;
    procedure Ligar(); override;
  end;

implementation

uses Dialogs;
{ TAviao }

procedure TAviao.Ligar();
begin
  // repare que não vai inherited aqui
  // pois não existe nada na classe base
  ShowMessage('Ligando o aviao '+Descricao);
end;

procedure TAviao.Mover();
begin
  inherited; // isso vai chamar o Ligar
  ShowMessage(Descricao+' está voando.');
end;

end.
```

Nota

Em uma situação real, por exemplo, poderíamos ter um cadastro base que tem um método polimórfico chamado Gravar. Sabemos que antes de Gravar dados no BD precisamos sempre Validar as informações, mas essa validação vai depender das classes concretas descendentes. Validamos um cliente de uma forma, um produto de outra, e assim por diante. Então, poderíamos ter uma classe base chamada TCadastro que possui um método virtual chamado Gravar, que sempre chama Validar, que por sua vez é abstrato. Nunca mais precisamos chamar o Validar, apenas implementar.

E na classe base, *TMeioTransporte*, simplesmente chamamos o *Ligar*:

```
procedure TMeioTransporte.Mover();
begin
  Ligar();
end;
```

Vamos executar a aplicação agora e ver como tudo isso vai funcionar. No form principal preencha os *Edits* e crie um carro. Veja o que acontece passo a passo:
1 – Ao clicar no botão *Mover*, será chamado o *Mover* da classe *TMeioTransporte*. Como esse método é *virtual*, ele identifica o método pela instância criada e não pelo tipo declarado, logo será chamado *Mover* de *TCarro*;
2 – *Mover* de *TCarro* faz uma chamada à *inherited* o que invoca o método de mesmo nome na classe base, *TMeioTransporte.Mover*;
3 – *TMeioTransporte.Mover* chama *Ligar* para ligar o meio de transporte;
4 – *Ligar* é um método *virtual / abstrato*, logo será chamado *TCarro.Ligar*;
5 – Após a chamada à *inherited* retornamos a *TCarro.Mover*, que então executa o restante do código.

Vamos ver um exemplo na VCL? Na classe *TDataSet*, achamos o seguinte método abstrato:

```
procedure InternalClose; virtual; abstract;
```

Ou seja, ele está apenas aí declarado, não vamos achar sua implementação em *TDataSet*, porém, ele pode ser chamado, como um comportamento comum, que mais tarde vai tomar diferentes formas. Ele é chamado no método *TDataSet.CloseCursor*, que é um método *virtual* (porém não abstrato).

Construtores e Destrutores

Um *construtor* é um método especial encarregado de alocar a memória para um objeto. Ao contrário, um *destrutor* libera a memória alocada para o objeto. Muitas vezes precisamos adicionar código personalizado a um construtor, por exemplo, para alocar memória para um objeto interno ou inicializar variáveis. Em nosso exemplo, vamos utilizar construtores para inicializar as variáveis do objeto. Declare o seguinte na classe *TCarro*:

```
constructor create();
```

Aperte *Shift+Ctrl+C* e implemente o construtor como mostrado na **Listagem 5.** Faça o mesmo para a classe *TAviao* (**Listagem 6**). Como você pode ver, usamos o construtor para inicializar atributos das classes. Podemos inicializar as variáveis comuns (como *Descricao* e *Capacidade*) no construtor da classe base *TMeioTransporte* (**Listagem 7**).

Listagem 5. Implementando um construtor: TCarro

```
constructor TCarro.create();
begin
  inherited; // chama o construtor da classe base
  Quilometragem := 0;
end;
```

Listagem 6. Implementando um construtor: TAviao

```
constructor TAviao.create();
begin
  inherited; // chama o construtor da classe base
  HorasVoo := 0;
end;
```

Listagem 7. Implementando um construtor: TMeioTransporte

```
constructor TMeioTransporte.create();
begin
 inherited;
 Capacidade := 0;
 Descricao := 'Sem Nome';
end;
```

Construtores inicializam objetos e são chamados quanto um objeto é instanciado com *Create* (ou *new* no *Delphi Prism*). Quando chamamos *free* de um objeto, ele é liberado, isso vai fazer uma chamada ao seu destrutor, oportunidade que temos para liberar objetos internos que alocamos em nossa classe. Para criar um destrutor, você deve declará-lo na interface da classe e implementá-lo como na **Listagem 8.**

Listagem 8. Implementando um destrutor

```
destructor destroy; override;
...
destructor TMeioTransporte.destroy;
begin
  // seu código de limpeza aqui
  inherited;
end;
```

Na VCL temos um exemplo clássico. Vamos examinar a classe *TMemo*, na verdade, *TCustomMemo* (**Listagem 9**). Note que um *Memo*, obviamente, armazena uma lista de strings (*TStrings*, que é uma classe abstrata). Logo, vemos o atributo privado *FLines* do tipo *TStrings*. Uma propriedade chamada *Lines* mapeia para o atributo privado interno. No construtor, *FLines* é inicializado, alocado em memória, e no destrutor, é destruído. Com isso, quando um *Memo* for destruído, ele mesmo se encarrega de liberar todos os seus objetos internos alocados, evitando *memory leaks* na aplicação, já que no Win32 não temos um coletor de lixo (*Garbage Collector*) como no .NET.

Nota

Observe que destroy é declarado como virtual em TObject, logo devemos dar um override ao sobrescrever. Ao contrário, o constructor é estático, e só passa a ser virtual a partir de TComponent.

Listagem 9. Exemplo de construtor / destrutor na VCL

```
TCustomMemo = class(TCustomEdit)
private
  FLines: TStrings;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  property Lines: TStrings read FLines write SetLines;
  ...
end;
...
constructor TCustomMemo.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
  FLines := TMemoStrings.Create;
end;
...
destructor TCustomMemo.Destroy;
begin
  FLines.Free;
  inherited Destroy;
end;
```

Nota do DevMan

Existem muitas classes TCustomXXX na VCL. Elas se encarregam de implementar toda a funcionalidade de uma classe, de um controle, porém, não publicam suas propriedades. Um exemplo clássico na VCL é o TCustomClientDataSet. Ele implementa toda a funcionalidade do TClientDataSet. Mas se observar o código-fonte de TClientDataSet, em dbclient.pas, verá que o tão popular e falado componente da VCL, sendo considerado por muitos (inclusive eu – não o DevMan, o autor) como o melhor componente da VCL, na verdade não faz nada! Ele apenas muda o especificador de visibilidade de propriedades declaradas e implementadas na classe custom base. Por exemplo, CommandText é todo implementado em TCustomClientDataSet e declarado como protected. TClientDataSet simplesmente publica a propriedade:

```
TClientDataSet = class(TCustomClientDataSet)
published
 property CommandText;
 ...
```

Especificadores de Visibilidade

Até aqui nossas classes utilizaram um único especificador de visibilidade para os atributos e métodos (*public*). Um atributo / método com esse modificador pode ser acessado (visto) por outras classes. Porém, temos inúmeros outros especificadores no *Object Pascal*, como podemos ver na **Listagem 10**.

Listagem 10. Especificadores de visibilidade / modificadores

```
type
  TMinhaClasse = class(TObject)
  private

  protected

  public

  published

  end;
```

Vejamos brevemente o que significa cada um:

Private – Membros definidos com esse especificador são visíveis somente na classe atual e classes "amigas" (*friendly* classes), ou seja, classes declaradas na mesma unit;

Protected - Visível somente na classe atual, descendentes e por classes amigas;

Public - Visível a partir de qualquer outra classe;

Published - Visível a partir de qualquer outra classe, ativando suporte à RTTI, com a principal finalidade de serem vistas no *Object Inpector*.

Voltando ao nosso exemplo, localize a declaração da classe *TMeioTransporte* e altere como mostrado na **Listagem 11.** Observe que foi colocada a letra "F" no início dos campos privados, para indicar que se tratam de *Fields*. Observe também que o método abstrato *Ligar* foi declarado na seção *protected*. Isso porque esse método não é chamado a partir de outra classe, é chamado internamente pela própria classe *TMeioTransporte* e é sobrescrito pelas classes descendentes. Por exemplo, se criarmos uma instância de *TMeioTransporte* e chamarmos o método *Ligar*, obteremos um *Abstract Error*.

O motivo de movermos as variáveis para a seção *private* foi encapsular o seu acesso. Porém, se tentarmos compilar a aplicação, vamos ter alguns erros, pois variáveis privadas não são acessíveis fora do escopo da classe. Criaremos então propriedades para permitir o acesso às variáveis privadas.

Propriedades

Adicione então o último especificador de visibilidade à nossa classe, declarando duas propriedades que fazem o mapeamento para os campos privados. Utilize a palavra-reservada *property* para definir uma propriedade, como na **Listagem 12**.

Listagem 11. TMeioTransporte, agora bem estruturada (porém sem propriedades)

```
TMeioTransporte = class
private
  FCapacidade : integer;
  FDescricao : string;
protected
  procedure Ligar(); virtual; abstract;
public
  constructor create();
  destructor destroy(); override;
  procedure Mover(); virtual;
end;
```

Listagem 12. Propriedades

```
...
published
  property Capacidade: integer read FCapacidade write
FCapacidade;
  property Descricao: string read FDescricao write FDescricao;
end;
```

Nota

Na prática, nunca instanciamos classes abstratas, mas sim classes concretas que herdam e implementam classes abstratas. Na VCL, por exemplo, você nunca vai instanciar um TDataSet (apesar de isso ser possível), você declara um TDataSet e dentro dele coloca um TClientDataSet, TSqlDataSet etc. Interfaces resolvem esse problema. Veremos interfaces adiante neste curso.

Altere a definição de *TCarro* e *TAviao*, criando propriedades e definindo os especificadores de visibilidade, como fizemos em *TMeioTransporte*. O código final das classes pode ser visto na **Listagem 13**. Note que já estou usando métodos *get / set*, tópicos que vamos discutir a seguir.

Métodos Get/Set

Uma grande vantagem de se usar propriedades ao invés de campos (*Fields*) é a possibilidade de usar métodos que fazem o mapeamento da atribuição e leitura das variáveis privadas. Por exemplo, na definição da classe *TCarro* temos o seguinte:

```
property Quilometragem: integer
  read GetQuilometragem write SetQuilometragem;
```

Ao implementar uma propriedade assim, é só apertar *Shift+Ctrl+C* para que o *Delphi* crie os cabeçalhos e implementação dos métodos *get / set*. A implementação pode ser vista na mesma **Listagem 13.** Observe que nossa rotina *Get* apenas repassa o valor para a variável privada, reforçando assim princípios de encapsulamento. Em nossa rotina *Set* testamos se o valor passado não foi negativo, e se for, configuramos a variável privada como zero. Observe o seguinte código:

```
Carro.Quilometragem := -3;
```

Isso fará ser disparado o método *Set*, e o valor da *Quilometragem* passará a valer 0.

Listagem 13. Código final das classes, com especificadores de visibilidade e propriedades

```
[TMeioTransporte]

unit uMeioTransporte;

interface

type
  TMeioTransporte = class
  private
    FCapacidade : integer;
    FDescricao : string;
  protected
    procedure Ligar(); virtual; abstract;
  public
    procedure Mover(); virtual;
  published
    property Capacidade: integer read FCapacidade write
FCapacidade;
    property Descricao: string read FDescricao write FDescricao;
  end;

implementation

{ TMeioTransporte }
uses
  Dialogs;

procedure TMeioTransporte.Mover();
begin
  Ligar();
end;

end.

[TCarro]

unit uCarro;

interface

uses
  // coloque essa unit p/ acessar a classe TMeioTransporte
  uMeioTransporte;

type
// observe que TCarro agora herda de TMeioTransporte
  TCarro = class(TMeioTransporte)
    // observe que retiramos os campos Capacidade e Descricao daqui
  private
    FQuilometragem : integer;
    function GetQuilometragem: integer;
    procedure SetQuilometragem(const Value: integer);
  protected
    procedure Ligar; override;
  public
    procedure Mover; override;
  published
     property Quilometragem: integer
       read GetQuilometragem write SetQuilometragem;
  end;

implementation

uses Dialogs;
{ TCarro }

function TCarro.GetQuilometragem(): integer;
begin
   result := FQuilometragem;
end;

procedure TCarro.Ligar();
begin
  // repare que não vai inherited aqui
  // pois não existe nada na classe base
  ShowMessage('Ligando o carro '+Descricao);
end;

procedure TCarro.Mover();
begin
  inherited;
  ShowMessage(Descricao + ' entrou em movimento.');
end;

procedure TCarro.SetQuilometragem(const Value: integer);
begin
   if Value < 0 then
    FQuilometragem := 0
 else
   FQuilometragem := Value;
end;

end.

[TAviao]
unit uAviao;

interface

uses
  // coloque essa unit p/ acessar a classe TMeioTransporte
  uMeioTransporte;

type
// observe que TAviao agora herda de TMeioTransporte
  TAviao = class(TMeioTransporte)
    // observe que retiramos os campos Capacidade e Descricao daqui
  private
    FHorasVoo: integer;
    function GetHorasVoo: integer;
    procedure SetHorasVoo(const Value: integer);
  protected
    procedure Ligar(); override;
  public
    procedure Mover(); override;
  published
    property HorasVoo: integer
      read GetHorasVoo write SetHorasVoo;
  end;

implementation

uses Dialogs;
{ TAviao }

function TAviao.GetHorasVoo(): integer;
begin
  result := FHorasVoo;
end;

procedure TAviao.Ligar();
begin
  // repare que não vai inherited aqui
  // pois não existe nada na classe base
  ShowMessage('Ligando o aviao '+Descricao);
end;

procedure TAviao.Mover();
begin
  inherited;
  ShowMessage(Descricao + ' está voando.');
end;

procedure TAviao.SetHorasVoo(const Value: integer);
begin
  if Value < 0 then
    FHorasVoo := 0
  else
    FHorasVoo := Value;
end;

end.
```

Nota do DevMan

Existe um grande discussão em torno do Published. Vamos pensar: todos os componentes que largamos em um Form / DM são automaticamente published para poderem usar RTTI e terem suas propriedades persistidas no DFM. Porém, isso fere princípios básicos da POO. Ou seja, obrigatoriamente, esses componentes, mesmo que não queiramos, são públicos e podem ser acessados de outras classes. Se movermos um ClientDataSet usado em design (DFM) para a seção private, por exemplo, para reforçar o encapsulamento, obteremos um erro (a não ser que o criemos todo em runtime). Por conta disso, encontrei na Internet no blog do time do Delphi pedidos para a implementação do modificador strict published, que resolveria esse problema. Aproveitando, nas versões mais recentes do Delphi já podemos encontrar modificadores strict private, strict protected, que têm o intuito de proteger o acesso a membros internos de uma classe a partir de classes amigas.

Uma curiosidade: existe uma prática muito usada na VCL por conta desse "recurso" de classes amigas. Se precisarmos, por exemplo, acessar um membro privado de um ClientDataSet a partir de um Form, podemos fazer algo como:

```
type
  THackCDs = class (TClientDataSet)
  end;

  TForm1 = class(TForm)
    ClientDataSet1: TClientDataSet;
    ...
  end;

...
  THackCDs(ClientDataSet1).AcessoAMembroPrivado
```

A técnica é simples, a classe hack está na mesma unit da classe do Form, logo, viram amigas. E portanto, podemos fazer um type-cast do CDS para a classe hack e acessar seus membros internos. POG.

Nota do DevMan

Automated surgiu no Delphi 2 e era originalmente usada para suportar OLE Automation / COM. Foi descontinuado pelo uso de Type Libraries para esse propósito. Veja meu artigo da edição 19 para mais informações sobre OLE e COM.

Nota do DevMan

O .NET Framework define mais alguns modificadores, como friendly (acessível para classes no mesmo pacote ou diretório) e internal (acesso limitado ao programa / assembly). Além disso, não há o conceito de published no .NET.

Conclusão

Nossas classes *TMeioTransporte*, *TCarro* e *TAviao* estão praticamente finalizadas. Estão agora usando na prática excelentes recursos da POO, como herança, polimorfismo e encapsulamento. Vimos como compartilhar dados e comportamento. Criamos propriedade, estudamos os especificadores de visibilidade, vimos que até mesmo a VCL tem alguns "defeitos" de modelagem por conta de restrições do *Object Pascal* do *Delphi Win32*. Aprendemos o que são construtores e destrutores, além de métodos *get / set*. Ainda tem mais! Na próxima parte deste curso vamos dar um grande salto usando algo que é marca registrada do *Delphi* desde a sua primeira versão: vamos transformar nossas classes em componente e colocá-las dentro de pacotes. Veremos ainda como usar herança visual de formulários e também um dos recursos mais fantásticos da OO, *interfaces*. E para finalizar com chave de ouro, um estudo de caso de uma aplicação real que vai mostrar na prática como usar OO, incluindo herança, interfaces e polimorfismo, reforçando tudo o que vimos até aqui. E futuramente, vamos ver que a OO também tem seus defeitos, e usando a própria OO encontramos soluções, através de *Design Patterns*. Até a próxima edição! ●

Nota

Veremos mais sobre propriedades, rotinas get/set e especificadores de visibilidade nas próximas partes desta série, quando criarmos um componente.

Links

Blog do autor

http://guintherpauli.blogspot.com e http://twitter.com/guintherpauli

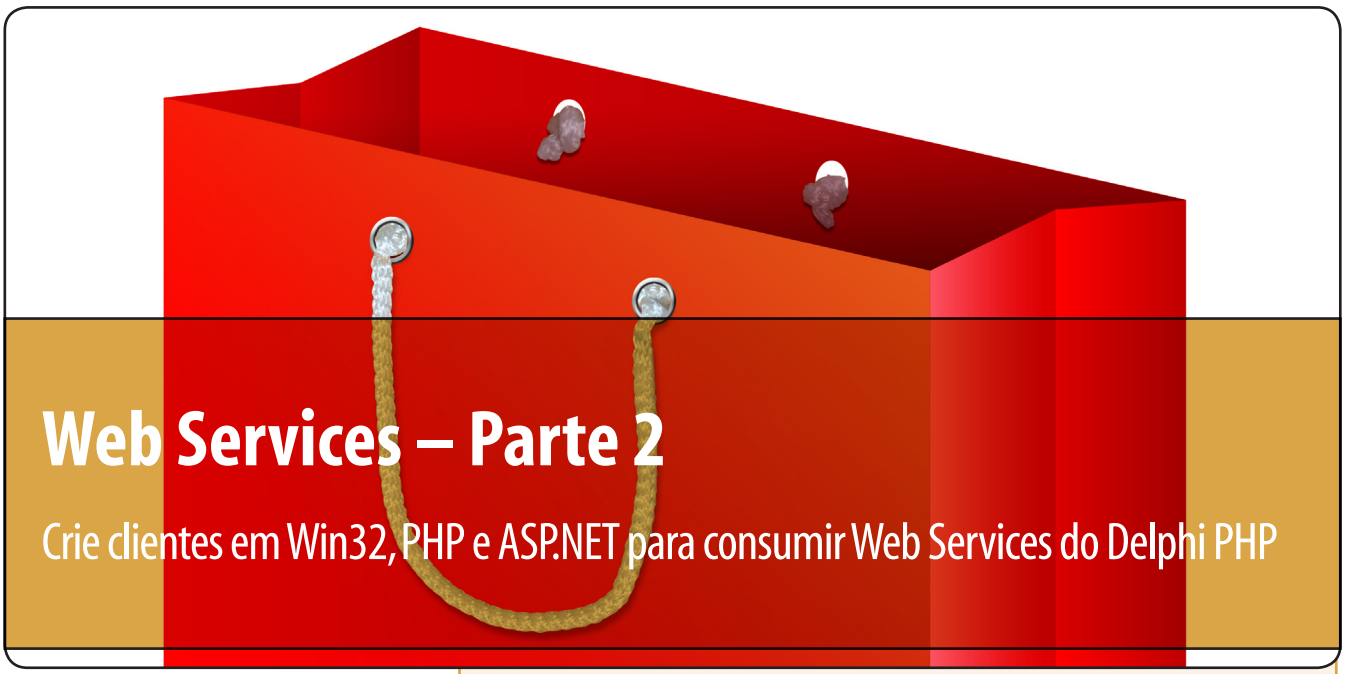
Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP



Web Services – Parte 2

Crie clientes em Win32, PHP e ASP.NET para consumir Web Services do Delphi PHP

Do que trata o artigo
Desenvolvimento de aplicações clientes para consumir Web Services. Essas aplicações serão desenvolvidas utilizando diferentes linguagens de programação e consumirão um mesmo Web Service desenvolvido na edição 107

Para que serve
Permitir que nós possamos oferecer a mesma solução para um cliente em diferentes formatos. Para empresas que optam por solução free oferecemos PHP rodando em Linux. Para uma empresa com infra Microsoft, como IIS, oferecemos uma solução em ASP.NET e para pequenas empresas podemos oferecer uma aplicação Win32

Em que situação o tema é útil
Num cenário onde temos que integrar diferentes tecnologias, como servidores escritos em PHP e clientes em ASP.NET. Hoje é muito comum as empresa portarem seus sistemas e serviços para Web. Com o uso de Web Services podemos criar diversos clientes para consumir o mesmo serviço de forma transparente

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br
Apaixonado por tecnologia da informação. Atualmente é sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria na área de TI e desenvolvimento de softwares customizados, parceira da TDS Tecnologia - RJ. Possui os títulos de BDS2006 Win32 Product Certified e Borland Instructor Certified. Editor Técnico da Revista Clube Delphi e Coordenador da área de Delphi no sistema de suporte on-line da DevMedia. Palestrante da Borland Conference 2007. Mantenedor da comunidade www.delphisophp.com, a primeira do Brasil 100% feita com Delphi for PHP. Autor de diversos artigos técnicos veiculados nos principais sites e revistas sobre Delphi do Brasil.

Resumo do DevMan

Na edição 107 mostramos como criar Web Services com Delphi for PHP. Criamos um pequeno Web Service com quatro métodos que acessam um banco de dados MySql. Neste artigo criaremos três aplicações clientes para consumir este Web Service. Assim a leitura do artigo da edição 107 se faz obrigatória para um perfeito entendimento do assunto, visto que toda a parte teórica foi abordada no artigo anterior.

Na edição 107 eu abordei o que são e como criar *Web Services* no *Delphi for PHP*. Terminamos o artigo com o *Web Service* pronto, porém não vimos como consumi-lo, isso será feito neste artigo. Eu não abordarei os detalhes técnicos e de funcionamento de um *Web Service*, pois isso já foi feito no artigo anterior. Assim vamos ficar apenas com a parte prática. Criaremos três aplicações, uma em cada tecnologia, para ver na prática as vantagens de se oferecer serviços através de *Web Services*.

Publicando o serviço

Antes de começarmos a desenvolver as aplicações para consumir nosso *Web Service*, publicarei o *Web Service* que desenvolvemos na edição passada em um provedor próprio que tenho com suporte à PHP. Assim ficará mais fácil o desenvolvimento dos exemplos, sem falar que você enquanto lê e desenvolve estes exemplos poderá consumir um serviço real hospedado na *Web*.

A primeira coisa que vou fazer é abrir o projeto do *Web Service* no *Delphi for PHP*. Feito isso vou trocar todas as referências que faço ao banco de dados local, ou seja, no meu código em todos os lugares em que conecto ao servidor *localhost* irei alterar para o endereço de um banco de dados que tenho hospedado no meu servidor *Web*. Meu servidor *MySql* fica em *mysql.rmfactory.com.br*. Por questões de segurança não irei informar aqui o usuário e senha para acesso à base de dados.

Feitas as alterações é hora de gerar os scripts para enviar ao servidor. Para isso eu acessei o menu *Tools>Deployment Wizard*. Essa ferramenta irá coletar todos os scripts do seu projeto e da *VCL* que seu projeto utiliza e colocará num diretório que você especificar. Observe o *Wizard* em operação na **Figura 1.**

Ao final, dentro do diretório especificado, estarão os seus scripts e uma pasta VCL. Nesta pasta estão os scripts necessários para o funcionamento da aplicação. Basta enviar o conteúdo do diretório para seu servidor *Web*. Eu o fiz e o *Web Service* está hospedado em *http://www.rmfactory.com.br/services/devmedia*.

Com o serviço já hospedado e rodando basta apenas consumi-lo. Se algum dia, alguém chegar para você e lhe disser: "*Teremos que construir um cliente para um Web Service*", então a primeira coisa que você irá precisar é o *WSDL*. O *Delphi Win32* e *Delphi .NET* já convertem o *WSDL* em um arquivo *.pas* com todas as classes necessárias para consumir os serviços. *WSDL* é a sigla de *Web Services Description Language*, padrão baseado em *XML*. Funciona como uma espécie de "*TypeLibrary*" do *Web Service*, além de ser usado para a validação das chamadas dos métodos (veja mais informações na Nota do DevMan). O endereço *WSDL* do nosso *WebService* está em *http://www.rmfactory.com.br/services/devmedia/WSDevMedia.php?wsdl*.

Criando o cliente Win32

Abra o *Delphi* e crie uma nova aplicação VCL. Salve a aplicação em um diretório de sua preferência. Renomeie a *unit1* para *uFrmPrincipal* e o projeto para *ClienteWin32*. A primeira coisa que vamos fazer é importar a *WSDL*. O *Delphi* traz um utilitário chamado *WSDL Importer* que faz o mapeamento da WSDL e gera uma *unit* com os métodos prontos para consumir os serviços, o que chamamos de classe *proxy*. Acesse o menu *File>New>Other* e no diálogo que se abre selecione *Delphi Projects>WebServices* e dê um duplo clique no item *WSDL Importer* (**Figura 2**). Informe o endereço do descritor de serviço, no nosso caso o endereço da *WSDL* que indiquei anteriormente (**Figura 3**).

Ao pressionar *Next* o utilitário mostrará um resumo dos métodos que estão descritos na WSDL e que serão importados para a *unit*. Pressione *Finish*. Uma *unit* será gerada e nela temos o código descrito na **Listagem 1**.

Nota do DevMan

WSDL (WebService Description Language) é uma especificação desenvolvida pelo W3C que permite descrever Web Services segundo um formato XML, independentemente dos formatos de mensagem e dos protocolos de rede que sejam usados. A WSDL descreve os serviços disponibilizados à rede através de uma semântica XML, de forma que toda a informação necessária para se chamar um sistema distribuído e o procedimento necessário para que esta comunicação se estabeleça já estejam também descritos. Enquanto o SOAP especifica a comunicação entre um cliente e um servidor (o protocolo), a WSDL descreve os serviços oferecidos.

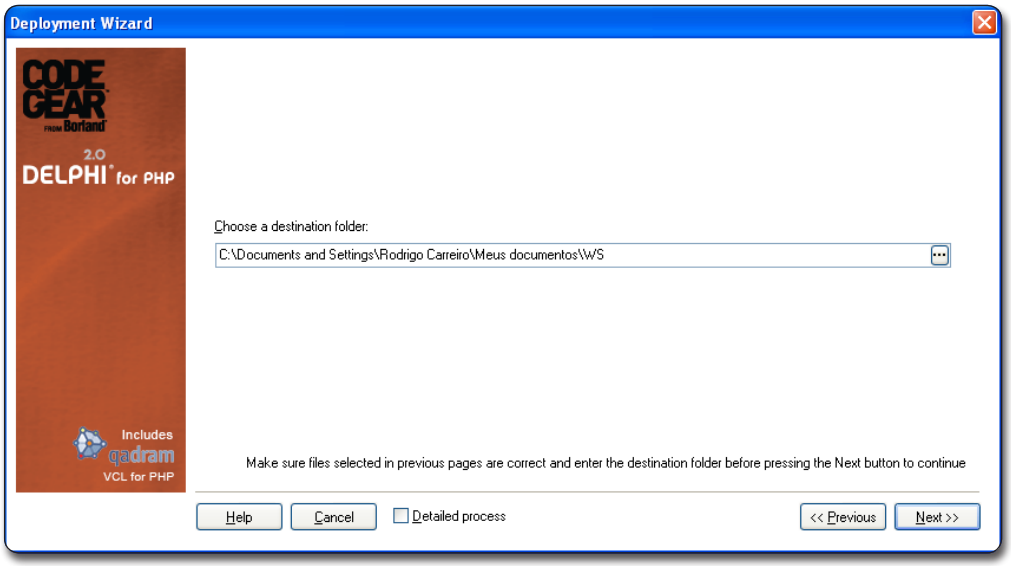


Figura 1. Deployment Wizard

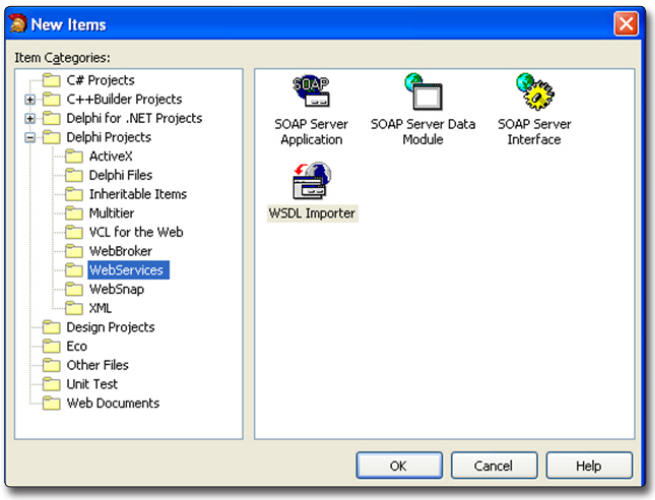


Figura 2. Acessando o WSDL Importer

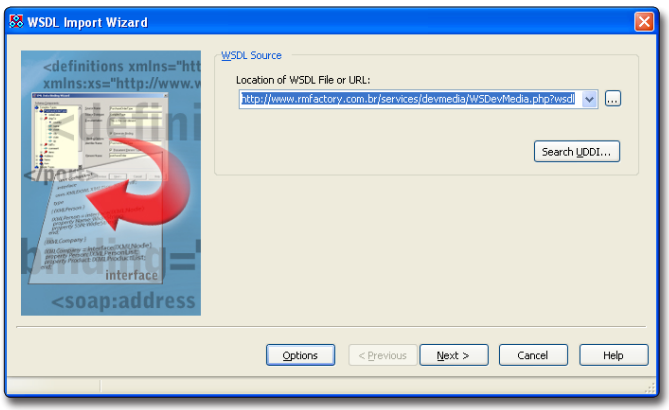


Figura 3. Importando a WSDL

Listagem 1. Código gerado pelo WSDL Importer

```
Web_Service_WorldPortType = interface(IInvokable)
  ['{677D3F52-BE48-FE65-7525-F166048CC560}']
    function  HelloWorld(const mesagem: WIDEString):
     WIDEString; stdcall;
    function  CountryByContinent(const Continent: WIDEString):
     WIDEString; stdcall;
    function  LanguagesByCountry(const Country: WIDEString):
     WIDEString; stdcall;
    function  GetSimpleCountry(const Country: WIDEString):
     WIDEString; stdcall;
  end;

function GetWeb_Service_WorldPortType(UseWSDL: Boolean=System.
False; Addr: string=''; HTTPRIO: THTTPRIO = nil): Web_Service_
WorldPortType;
…
```

No código da **Listagem 1** foram omitidos os comentários que por sinal possuem algumas informações úteis como *namespaces*, *url* etc. O que o utilitário fez na verdade foi criar uma *interface* com os métodos contidos em nosso *Web Service*. Entenda esta *interface* como um "controle remoto", a forma que usamos para efetuar a comunicação com o servidor. Uma função também é criada e tem por objetivo criar um objeto *HTTPRio*, que será responsável por conectar ao *Web Service* e invocar o método especificado.

Desta forma o acesso ao servidor fica transparente. Basta utilizar a função *GetWeb_Service_WorldPortType* que retorna um ponteiro para a *interface Web_Service_WorldPortType*. Com isso temos acesso aos métodos e podemos consumir os serviços. Vamos aqui consumir apenas um método, pois a maneira será a mesma para acessar os demais. Vamos utilizar o método *GetSimpleCountry* como exemplo. Vale lembrar que este método retorna as informações de um determinado país, informado no parâmetro *Country*. O retorno da função é uma *String*, mas não esqueça que esta *string* é o conteúdo de um arquivo *XML*. Então como exibir os dados que serão retornados no *XML*? Podemos mapeá-lo através do componente *XMLDocument* ou então utilizar o já conhecido *ClientDataSet*.

O problema é que o *XML* utilizado pelo *ClientDataSet* é do tipo *DataPacket*, um tipo específico usado pelo *DataSnap*. O nosso *XML* não possui esse formato, então para que possamos exibir os dados em um *DBGrid*, por exemplo, teremos que transformá-lo em um *DataPacket* para que o mesmo possa ser entendido por um *ClientDataSet*. Aqui entra em cena o *XML Mapper*.

A função do *XML Mapper* é gerar um arquivo de transformação para que possamos converter um arquivo XML em um *DataPacket* e vice e versa. Não entrarei em detalhes sobre o uso do *XML Mapper* por não ser escopo deste artigo.

Para que possamos gerar o arquivo de transformação temos que ter em mãos o arquivo XML que queremos converter. Então somente em caráter excepcional vamos consumir um método do *Web Service* apenas para gerar um arquivo *XML*. Insira,um botão no form e no evento *onClick* digite o código da **Listagem 2.**

O código é bem simples. Nós temos uma variável do tipo *TStringList* na **linha 2**. Este objeto será usado apenas para que possamos usar o método *SaveToFile* (**linha 6**) e gerar o

Nota do DevMan

Em algumas linguagens de programação, o termo interface é uma referência à característica que permite a construção de aplicações que isolam do mundo exterior os detalhes de implementação de um componente de software. Um exemplo clássico de utilização de interfaces é o do sistema operacional que, através de uma interface de programação de aplicativos (API), permite que os programas utilizem os recursos do sistema (memória, CPU etc.) sem que os seus detalhes de implementação sejam conhecidos do programador. Este esquema isola e protege o sistema operacional de eventuais erros cometidos pela aplicação.

Nota do DevMan

TXMLDocument é um componente disponível na VCL do Delphi que implementa as interfaces do modelo DOM para manipulação de arquivos XML. Com ele é possível tanto ler como gerar arquivos XML.

Nota do Editor

A edição 36 apresenta um artigo completo sobre XML Mapper, intercâmbio de dados em XML e aplicações B2B (Business to Business).

Nota do DevMan

O DataPacket é uma estrutura utilizada internamente pelo ClientDataSet. Está estrutura é organizada em formato XML e é este DataPacket que é trafegado entre o Cliente e o Servidor de aplicação em projetos desenvolvidos em camadas.

Listagem 2. Gerando o arquivo XML

```
1 var
2  S: TStringList;
3 begin
4  S := TStringList.Create();
5  S.Add(GetWeb_Service_WorldPortType.GetSimpleCountry ('Brazil'));
6  S.SaveToFile('country.XML');
7  FreeAndNil(S);
8 end;
```

arquivo. Na **linha 5** invocamos o método *GetSimpleCountry* passando um país. Este método retorna um conteúdo do tipo *WideString* que na verdade é o conteúdo do nosso *XML*. Nós apenas colocamos este conteúdo dentro do objeto *TStringList* e o salvamos como *country.xml*. Dentro do IDE acesse o menu *Tools>XML Mapper*. Será aberto o utilitário mostrado na **Figura 4.**

Na tela principal do *XML Mapper* acesse o menu *File>Open* e abra o arquivo *country.xml* que acabamos de gerar. Ao executar este procedimento, o *XML Mapper* exibirá os nós que estão contidos no XML. Clique com o botão direito sobre qualquer um dos nós e selecione a opção *Select All*. Isso fará com que todos os nós sejam mapeados e fiquem prontos para serem transformados. Ao realizar este procedimento, o *XML Mapper* deverá estar como a **Figura 5**.

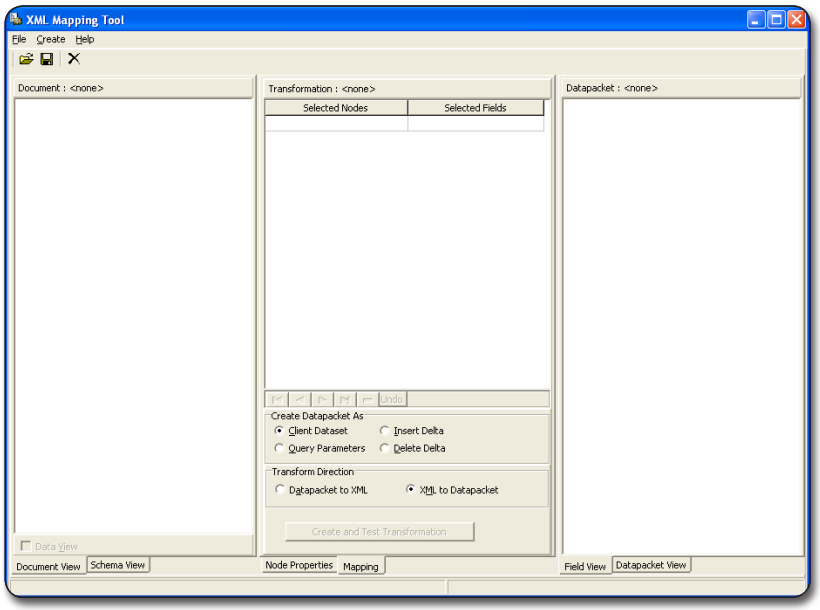


Figura 4. XML Mapper

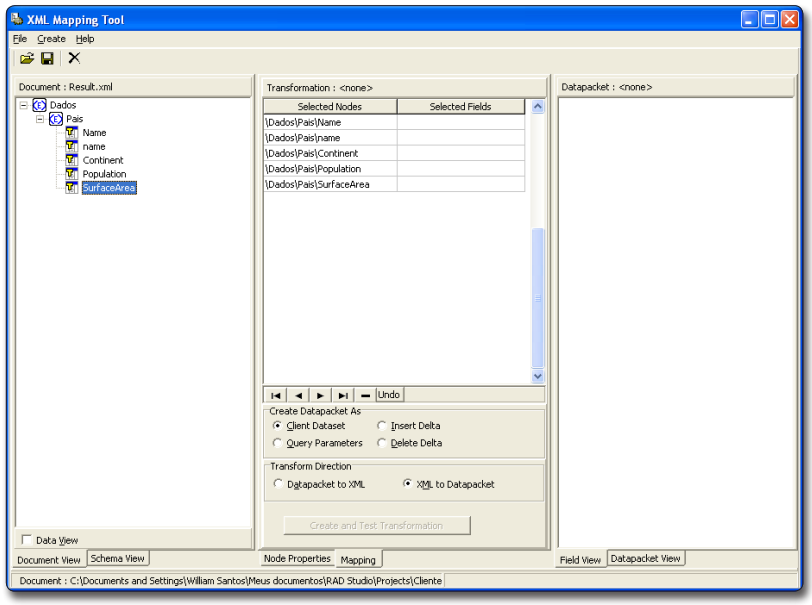


Figura 5. Arquivo XML carregado

Feito isso vamos gerar a transformação. Para isso acesse o menu *Create>DataPacket From XML*. Ao realizar este procedimento, o *XML Mapper* mostra do lado direito a estrutura do *DataPacket* **(Figura 6).**

Para finalizar o processo vamos salvar um arquivo *.xtr* para que possamos realizar esta transformação em tempo de execução. Acesse o menu *Create>Transformation* e a seguir *File>Save> Transformation*. Salve o arquivo *ToDp.xtr* no mesmo diretório da aplicação.

No formulário principal da aplicação vamos adicionar os seguintes componentes: *ClientDataSet*, *DataSource*, *DBGrid*, *Edit*, *Button* e *XMLTransform*. Conecte o *DBGrid* ao *DataSource* e o *DataSource* ao *ClientDataSet*. Posicione o botão ao lado do *Edit* e os dois acima do *DBGrid*. A **Figura 7** mostra um exemplo de como deverá estar seu formulário.

A função do componente *XMLTransform* é exatamente transformar o *XML* que virá do nosso *Web Service* para o formato *DataPacket*, para que este possa ser lido e carregado no *ClientDataSet*. Vá ao evento *OnClick* do botão e coloque o código da **Listagem 3.**

Listagem 3. Transformando o XML com XMLTransform

```
var
  S: String;
begin
  S := GetWeb_Service_WorldPortType.GetSimpleCountry(Edit1.Text);
  ClientDataSet1.XMLData := XMLTransform1.TransformXML(S,'todp.xtr');
end;
```

Primeiro invocamos o método *GetSimpleCountry* e passamos para o mesmo o nome do país informado no *Edit1*. O retorno desta função é guardado numa variável chamada *S*.

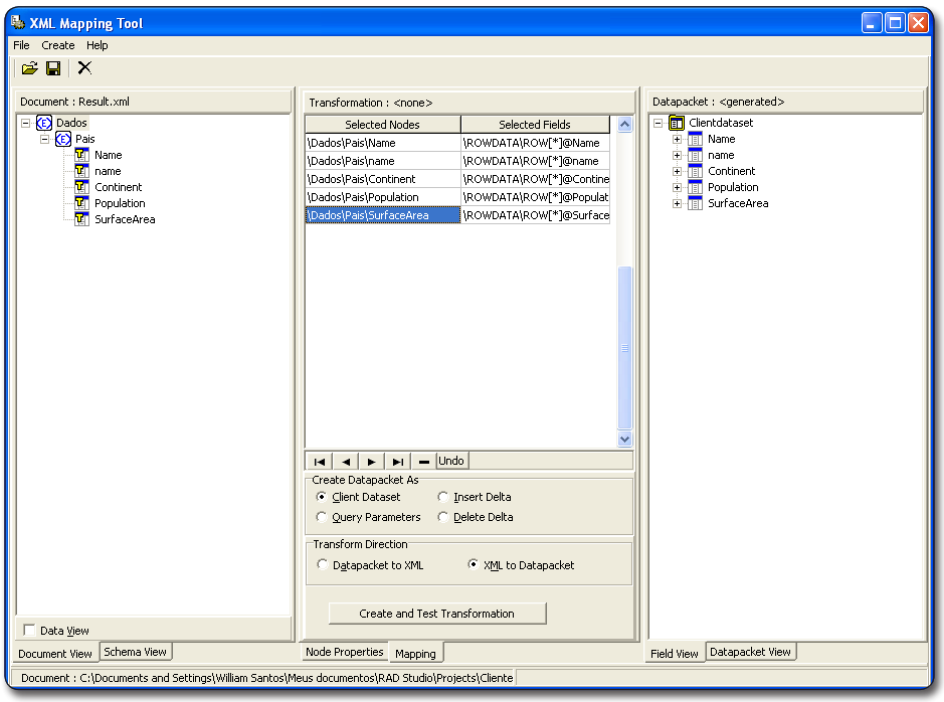


Figura 6. DataPacket gerado

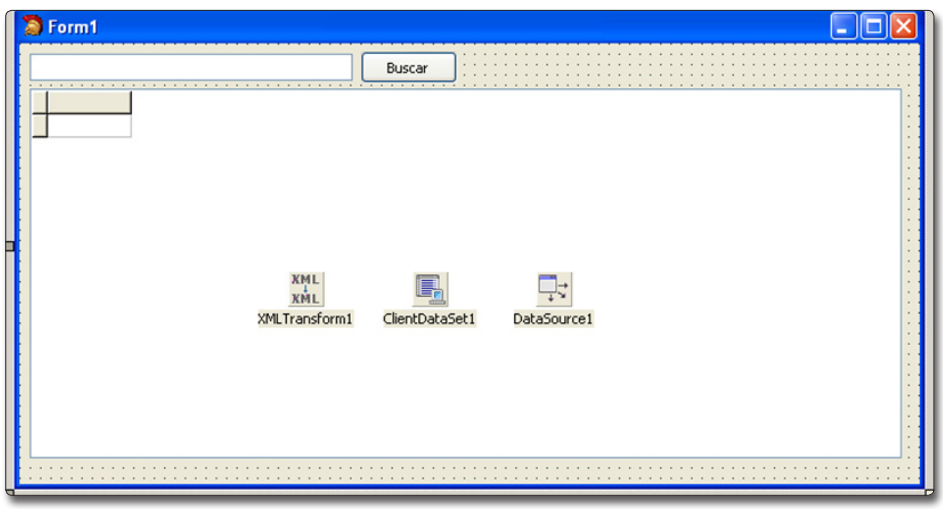


Figura 7. Formulário principal

Lembre-se que o conteúdo desta variável é na verdade um arquivo XML retornado pelo nosso *Web Service*. Feito isso nós utilizamos o componente *XMLTransform* e chamamos o método *TransformXML*, passando para ele a variável *S* e o nome do arquivo *.xtr* que criamos com o *XML Mapper*. Este método retornará um *DataPacket* que nós atribuiremos à propriedade *XMLData* do *ClientDataSet*. Rode a aplicação, digite o país *Brazil*, por exemplo, e observe o resultado (**Figura 8**).

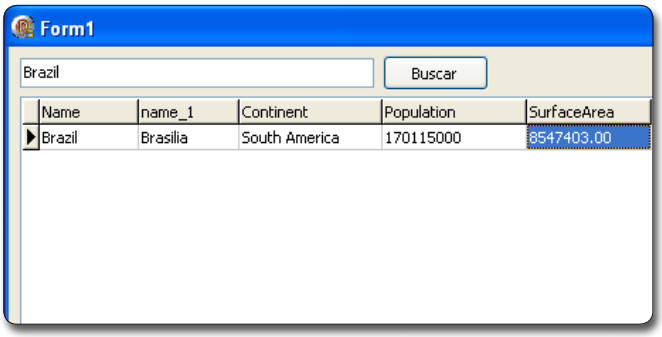


Figura 8. Aplicação em execução

Criando o cliente ASP.NET

No *RAD Studio* crie uma nova aplicação *ASP.NET*. No diálogo que se abrir informe o nome do projeto como *ClienteAspNet*. Seu projeto já traz uma página *Web* e nela vamos adicionar um componente *TextBox*, um botão e um componente *GridView*. Disponha os componentes conforme a **Figura 9.**

Interface criada vamos agora importar o *WSDL* para nossa aplicação. Para isso vá até o *Project Manager* e clique com o botão direto sobre o item *ClientAspNet.dll*. No menu que vai se abrir clique em *Add Web Reference*. Isso abrirá um utilitário que é mostrado na **Figura 10.**

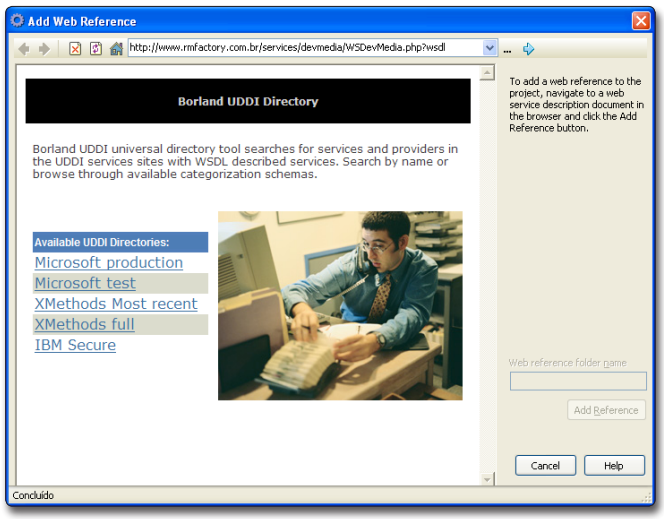


Figura 10. Importando a WSDL em .NET

Informe o endereço do *WSDL* que você quer importar, no nosso caso o mesmo utilizado na aplicação *Win32*. Clique no botão com a figura de uma seta azul para que seja gerado o código para consumirmos nosso *Web Service*. Em *Web Reference Folder Name* digite *WSDLRef* e clique no botão *Add Reference*. Repare que no *Project Manager* foi adicionada uma nova *unit*. A **Figura 11** exibe a nova estrutura.

O arquivo gerado em .NET é relativamente maior do que o gerado em *Win32*. Isto porque em .NET temos a opção, entre outras coisas, de fazer chamadas assíncronas ao métodos do nosso servidor *Web*. Por isso, para cada método no *Web Service*, temos três ou mais funções para consumi-lo.

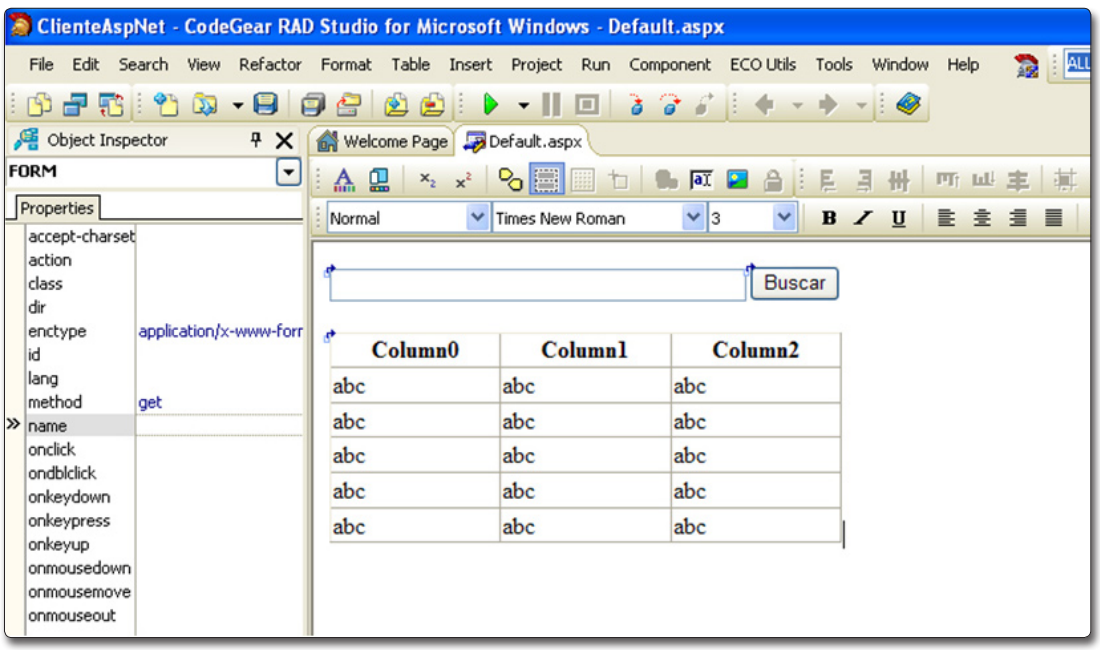


Figura 9. Interface da aplicação ASP.NET

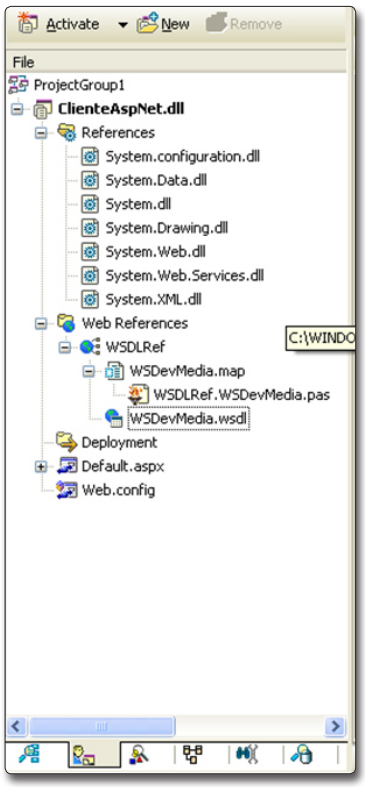


Figura 11. Web References

Dê um duplo clique no botão que está na nossa página ASP.NET e no evento *Click* insira o código da **Listagem 4.**

Repare que em vista do código que escrevemos em *Win32* este código em .NET está com algumas linhas a mais. Porém não houve em .NET a necessidade de se criar um arquivo de transformação. Isto porque a maioria dos objetos em .NET já são 100% compatíveis com fontes de dados em XML.

Listagem 4. Consumindo o serviço no .NET

```
1.  var
2.   WS: WebServiceWorld;
3.   Stream: StringReader;
4.   XmlReader : XmlTextReader;
5.   DS: DataSet;
6.  begin
7.   Ws := WebServiceWorld.Create();
8.   Stream := StringReader.Create(WS.CountryByContinent(TextBox1.Text));
9.   XmlReader := XmlTextReader.Create(Stream);
10. Ds := DataSet.Create();
11. Ds.ReadXml(XmlREader);
12. GridView1.DataSource := DS;
13. GridView1.DataBind();
14. end;
```

Começamos com um objeto do tipo *WebServiceWorld* que instanciamos na **linha 7**. Estes objetos (*proxy*) possuem todos os métodos que estão disponíveis em nosso servidor web. Na **linha 8** nós criamos um objeto do tipo *StringReader*. Ele irá fazer um *stream*, ou seja, colocará na memória o retorno da função que iremos chamar no *Web Service*, neste caso o retorno da função *CountryByContinent*. Então ao clicarmos no botão, iremos invocar o método *CountryByContinent* através do objeto *WebServiceWorld* e o seu retorno passamos para o objeto *StringReader*. **Na linha 9** instanciamos um objeto *XmlTextReader* responsável por ler documentos XML. No nosso caso estamos lendo o conteúdo do nosso *stream* e já o preparando para fazer *bind* no *GridView* (ou seja, exibir os dados). Na **linha 10** criamos um objeto *DataSet*, que mantêm uma estrutura de dados em memória. Ao criarmos o *DataSet* chamamos o método *ReadXML* para carregar o conteúdo *XML* que está na variável *XmlReader*. Dados carregados apenas passamos o *DataSet* para a propriedade *DataSource* do *GridView* e os dados já serão exibidos (**Figura 12**).

Criando o cliente PHP

Chegamos à última das aplicações clientes que vamos criar neste artigo e para finalizar iremos desenvolver uma

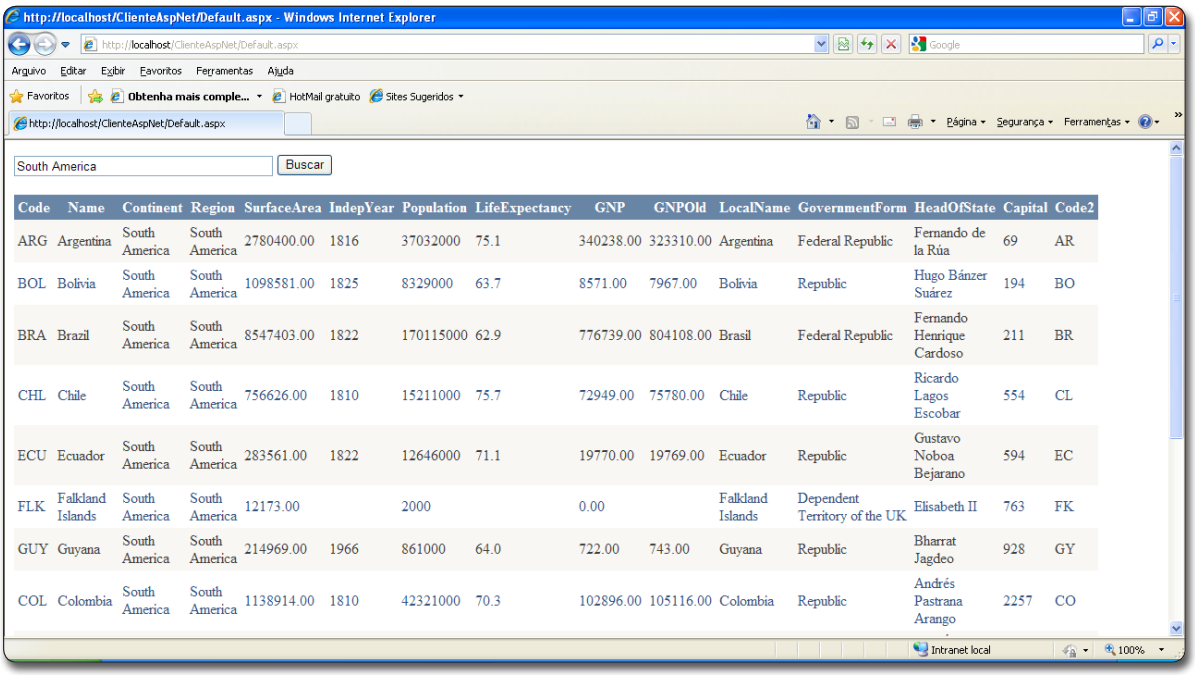


Figura 12. Aplicação ASP.NET em execução

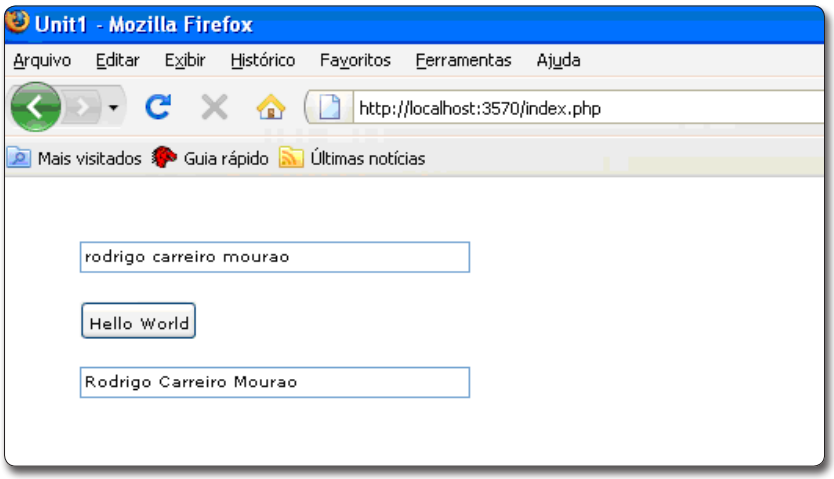


Figura 13. Aplicação cliente em PHP

pequena aplicação com *Delphi for PHP* para consumir um método do nosso *Web Service*. Então vamos abrir o Delphi for PHP e através do menu *File>New>Application* criar uma nova aplicação. Salve-a e no formulário adicione um botão e dois *Edits*. Aqui vamos invocar o método *HelloWorld* presente em nosso *Web Service*.

Para consumir um *Web Service* em *PHP* temos que fazer uso de uma classe chamada *nusoap*. Ela possui os métodos necessários para essa integração. Assim a primeira coisa que faremos é incluir uma referência ao script *nusoap.php*. Para isso adicione a seguinte linha na seção *uses* do formulário: *require_once('vcl/nusoap/nusoap.php')*. Agora no evento *onClick* do botão insira o código da **Listagem 5.**

Listagem 5. Consumindo o Web Service em PHP

```
$wsdl = "http://www.rmfactory.com.br/services/devmedia/
 WSDevMedia.php?wsdl";
$client = new nusoapclient($wsdl, true);
$parameters = array('mesagem' => $this->Edit1->Text);
$this->Edit2->Text = $client->call('HelloWorld', $parameters);
```

No código da **Listagem 5** criamos um objeto *nusoapclient* passando para ele o endereço da *WSDL*. Feito isso criamos um *array* para passar o parâmetro solicitado pela função *HelloWorld*. Na sequência utilizamos o método *Call* para invocar o método *HelloWorld* e passamos o *array* com o parâmetro. O retorno da função será a *string* passada formatada com a primeira letra de cada palavra em maiúscula. A **Figura 13** mostra a aplicação em execução.

Conclusão

Seja em *Win32*, PHP ou ASP.NET ficou claro que desenvolver aplicações integradas a *Web Service* é algo muito interessante, útil e necessário, tanto do ponto de vista técnico quanto financeiro. O assunto abordado neste e no anterior orientou você a oferecer serviços web e integrar suas aplicação com as dos seus clientes e fornecedores. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem momentos onde a estrutura faz toda a diferença.

Se o tempo é ruim, você encontra na Alog um porto seguro. Isto porque sua empresa reduz custos e melhora a governança em TI. Entre em contato e descubra porque a Alog é líder em hosting gerenciado.

REDUÇÃO DOS CUSTOS AUMENTO DA GOVERNANÇA EM TI

0800 282 3330 | alog.com.br

ALOG
DATA CENTERS DO BRASIL

HOSTING GERENCIADO | COLOCATION | CONTINGÊNCIA | E-MAIL CORPORATIVO | CONECTIVIDADE | SERVIÇOS PROFISSIONAIS

VOCÊ levou anos para desenvolver, testar e colocar seu produto no mercado e **ELE** apenas alguns minutos para **PIRATEÁ-LO.**

SafeNet tem à sua disposição a proteção de software mais avançada e segura. As Chaves de Hardware Sentinel oferecem tecnologia que incorpora criptografia de chave pública, criptografia AES e autenticação interna para garantir o mais alto nível de proteção anti-pirataria.

Peça seu Kit de Desenvolvimento hoje mesmo.

Mais informações www.safenet-inc.com/antipirataria/01
e receba o Estudo de Caso.

SafeNet®
The Foundation of Information Security